Haptic Teleoperation of Mobile-Manipulator Systems using Virtual Fixtures

by

Michael Wrock

Bachelor of Engineering (Hons), University of Ontario Institute of Technology, 2008

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Applied Science

in

The Faculty of Engineering and Applied Science

Mechanical Engineering

Supervisor(s):	Scott Nokleby, Faculty of Engineering and Applied Science
Examining Board:	Remon Pop-Iliev, Faculty of Engineering and Applied Science
External Examiner:	Andrew Hogue, Faculty of Business and Information Technology

THE UNIVERSITY OF ONTARIO INSTITUTE OF TECHNOLOGY

November 2011

Abstract

In order to make the task of controlling Mobile-Manipulator Systems (MMS) simpler, a novel command strategy that uses a single joystick is presented to replace the existing paradigm of using multiple joysticks. To improve efficiency and accuracy, virtual fixtures were implemented with the use of a haptic joystick. Instead of modeling the MMS as a single unit with three redundant degrees-of-freedom (DOF), the operator controls either the manipulator or the mobile base, with the command strategy choosing which one to move.

The novel command strategy uses three modes of operation to automatically switch control between the manipulator and base. The three modes of operation are called near-target manipulation mode, off-target manipulation mode, and transportation mode. The system enters near-target manipulation mode only when close to a target of interest, and allows the operator to control the manipulator using velocity control. When the operator attempts to move the manipulator out of its workspace limits, the system temporarily enters transportation mode. When the operator moves the manipulator in a direction towards the manipulator's workspace the system returns to near-target manipulation mode. In off-target manipulation mode, when the operator moves the manipulator to its workspace limits, the system retracts the arm near to the centre of its workspace to enter and remain in transportation mode. While in transportation mode the operator controls the base using velocity control.

Two types of virtual fixtures are used, repulsive virtual fixtures and forbidden region virtual fixtures. Repulsive virtual fixtures are present in the form of six virtual walls forming a cube at the manipulator's workspace limits. When the operator approaches a virtual wall, a repulsive force is felt pushing the operator's hand away from the workspace limits. The forbidden region virtual fixtures prevent the operator from driving into obstacles by disregarding motion commands that would result in a collision.

The command strategy was implemented on the Omnibot MMS and test results show that it was successful in improving simplicity, accuracy, and efficiency when teleoperating a MMS.

Dedication

To my family and friends, for your support and faith in my success.

Acknowledgements

First and foremost I would like to thank Dr. Scott Nokleby, without his knowledge, assistance, and guidance none of this would have been possible. He was an invaluable asset throughout the design, implementation, and experimentation of my research. I appreciate his guidance and understanding, allowing me to undergo my research in a way that best suited my learning style.

I would like to thank my family, who have always supported and believed in me. Without them I would not have the confidence or ambition to perform my very best.

I must also thank my friends and peers, particularly those who worked with me in the MARS lab. You were a much needed resource when I needed technical support.

Finally, I thank Cameco Corporation (through the Cameco Research Chair in Nuclear Fuel at UOIT), the Natural Sciences and Engineering Research Council (NSERC) of Canada, and the Canada Foundation for Innovation (CFI) for providing funding to support this research.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Both processing power and technical capabilities of robots and manipulators have increased at an accelerating rate in recent years, so much so that command strategies for these systems cannot fully exploit their capabilities in a way that maximizes control for the operator in an intuitive way. Much of the focus when it comes to controlling manipulator systems lies in autonomous or semi-autonomous control of the manipulator. When dealing with redundant systems, researchers favor a mathematical approach to resolving redundancy, often neglecting certain aspects of the system that can be used to its advantage. These mathematical solutions can lead to algorithmic singularities, increased processing requirements, or less than optimal performance. This work approaches the control of a mobile-manipulator system with the intention of direct control from an operator, i.e., teleoperation. The challenge then becomes identifying and implementing the most intuitive and simple to use command strategy for a mobile-manipulator system.

1.1 Mobile-Manipulator Systems

Manipulators can be used for a variety of purposes and are controlled in a variety of ways ranging from completely manual to fully autonomous. Control in its most basic form re-

quires an operator to individually control each degree-of-freedom (DOF) of the manipulator. Advanced control of manipulators can be done autonomously either through preprogrammed operations or using artificial intelligence. Current research strives to reach a level of artificial intelligence where a manipulator can perform a variety of tasks using its own problem solving abilities. Industry driven research uses autonomous mobile-manipulator systems to automate assembly and manufacturing tasks [1–3]. While robotic manipulators can perform some tasks using artificial intelligence, others require a human's reasoning, comprehension, and problem solving abilities.

When a human operator controls a manipulator, it is called "teleoperation" meaning operation at a distance. Teleoperated manipulators have two fundamental features required to perform their given task: an input device and an end-effector. The end-effector, located on the last link of the manipulator, holds the tool for the required task (see Figure 1.1). The end-effector can be a drill, welder, gripper, or even simply a pointing device. The input device is used to input the operator's desired end-effector position or motion.

While manipulators can come in many different shapes and sizes (as shown in Figures 1.2 to 1.5), from serial to parallel configurations, gantry to anthropomorphic, or nano-scale to macro-scale they all have the same drawback: they are fixed. A manipulator's workspace, defined as the summation of all reachable points, is finite due to limitations in link lengths. Workspaces can be expanded by increasing the size of the manipulator, but with increased size comes increased power consumption and cost. The best way to expand a manipulator's workspace is to attach it to a mobile robot. Though the manipulator's workspace is still limited, it can now be repositioned anywhere the mobile robot can travel, thus expanding the workspace to include all points the manipulator can reach through the repositioning of the base. A Mobile-Manipulator System (MMS) consists of these two devices connected together; the manipulator often referred to as the "arm" and a mobile robot usually referred

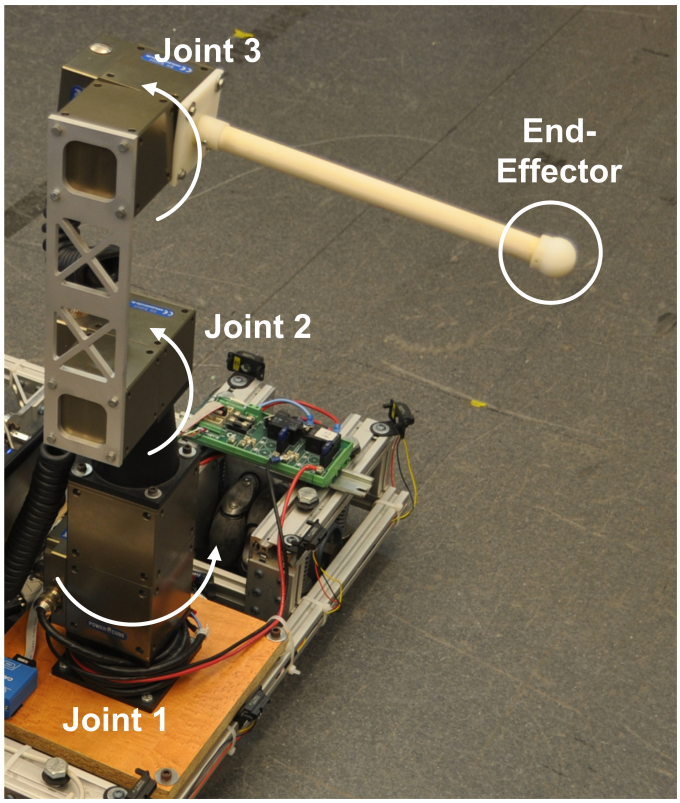


Figure 1.1: Three Links and End-Effector of a Serial Manipulator

to as the “base”.

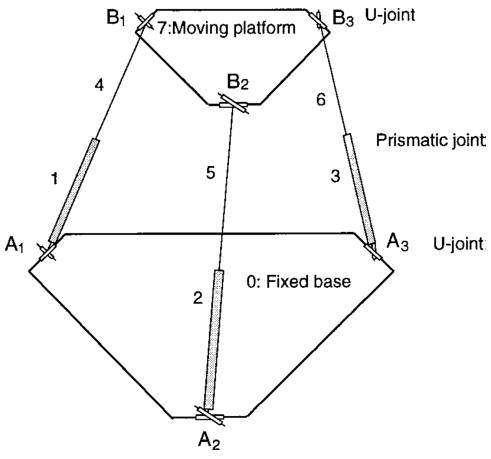


Figure 1.2: A Parallel Manipulator Configuration [4]

The benefits of a MMS versus a fixed manipulator system does not end with the increased workspace. Having one manipulator on a mobile base and multiple end-effector tools, a sin-

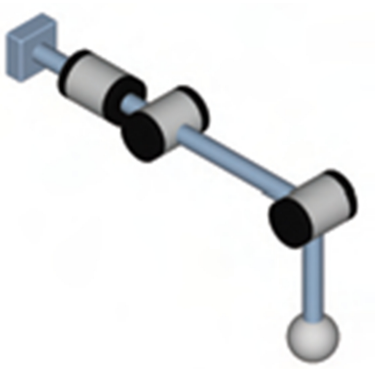


Figure 1.3: A Serial Manipulator Configuration [5]

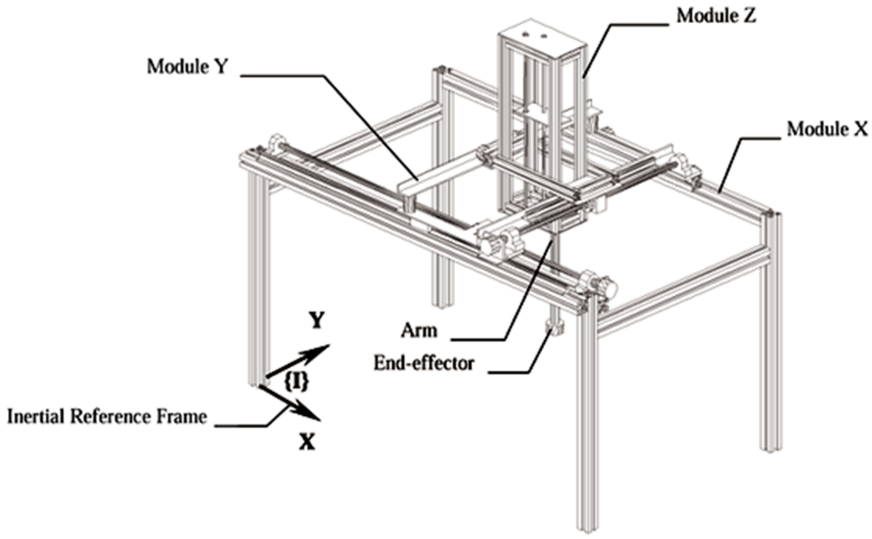


Figure 1.4: A Gantry Style Manipulator [6]

gle MMS can replace many fixed manipulators. Another significant advantage of a MMS is the ability to exploit its redundancies to maximize performance. Redundancies can be exploited to improve force capabilities and manipulability, as shown in Figures 1.6 and 1.7.

A redundant system has more DOF than is required for a specific task. For most tasks, six is the maximum DOF required since it allows full control over position and orientation. Sometimes a manipulator's strongest pose requires aligning two links along the same axis but in doing so puts the manipulator in a singular configuration, losing a DOF. By using the additional DOF from the base, the MMS is able to maintain the singular pose without

Figure 1.5: An Anthropomorphic Manipulator [7]

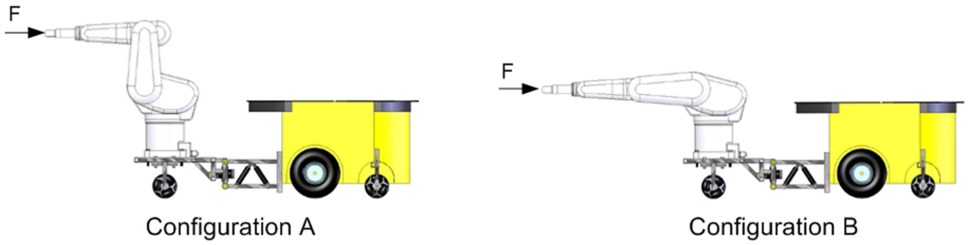


Figure 1.6: Exploiting Redundancy to Improve Force [8]

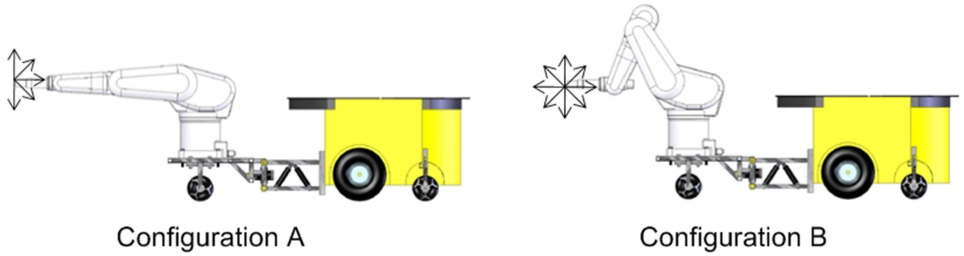


Figure 1.7: Exploiting Redundancy to Improve Manipulability [8]

losing a DOF. Another example of exploiting redundancy is collision avoidance. If the arm is attempting to manipulate an object but cannot do so without a collision, the base can maneuver in such a way to allow the arm unobstructed access.

There are, however, a few disadvantages of using MMS, such as the need for redundancy resolution, various mechanical drawbacks, and increased complexity. Redundancy reso-

lution is by no means a recent problem, and much research has gone in to the best way to resolve redundancies. There are many approaches to redundancy resolution, each with their own strengths and weaknesses. Along with redundancy resolution, one must consider the mechanical problems that arise with MMS. When in motion, there is typically increased mechanical vibrations caused by the base. As well, stability becomes an issue if the arm is much larger or heavier than the base and may cause it to tip over when fully extended. MMS are typically more expensive and complex than a single fixed manipulator, so in some cases it may be more reasonable to have multiple fixed manipulators than a single mobile manipulator.

1.2 Haptic Devices

The word haptic, from the Greek word "haptikos" means pertaining to the sense of touch. A haptic device is for touch what a speaker is for sound. A haptic device however is a two way information device, so a more accurate comparison would be a speaker and microphone. Haptic devices convey information through the sense of touch in the form of motions, forces, and/or vibrations. An example of a haptic device is the servo controlled operating "stick" in modern aircraft. Initially, the control of the aircraft was directly linked to the airplane's control surfaces, and operators would receive useful feedback from the wind forces over the ailerons such as wind buffeting when approaching a stall. When using servo-mechanisms to control the airplane, this feedback is not present so a vibration device was installed to simulate the wind buffeting sensation when approaching a stall. This is an example of haptic feedback, but haptic information can be much more detailed than that. When implemented on a spatial joystick (a joystick with the ability to move in Cartesian coordinates), haptic forces can be used to simulate any object virtually. Using the joystick, the operator can interact with a virtual object having mass, inertia, density, surface textures,

and many dynamic properties.

Haptic devices are usually designed in one of two ways. While the internal mechanisms may vary, haptic devices usually take the form of some kind of manipulator device, parallel or serial. One way to design a haptic device is to control the links using servos or motors and install a force and torque sensor to measure the forces at the end-effector. To control this device, the operator applies a force to the end-effector and to simulate the virtual object the haptic device moves accordingly. This kind of haptic is called an admittance haptic device. The second style of haptic device is called an impedance type device. Impedance devices use encoders to measure the position of the end-effector, and use backdriveable motors to apply forces that display haptic information.

1.3 Virtual Fixtures

Haptic devices are often used in teleoperation of manipulator systems. One style of teleoperation called bilateral teleoperation in its simplest form is two manipulators of the same size and configuration electronically linked [9]. One is called the master and the other the slave. The operator controls the master device, and the slave performs the same motions as the master. In this way, the operator feels like they are present at the slave's work site. This feeling is called "telepresence" and the more the master feels like the slave, the stronger the telepresence [10]. Without a strong telepresence the operator may accidentally drive the slave manipulator into another object. If the system is not backdriveable, the operator could cause considerable damage to the slave manipulator. Without backdriveability virtual fixtures can be used to avoid such a situation, and in fact backdriveability can be considered a virtual fixture.

Virtual fixtures are perceptual overlays of abstract information placed in the manipulator's

workspace. Since the overlays are hard to visualize, the virtual fixture metaphor was introduced by Rosenberg in 1993 [11]. Virtual fixtures come in many varieties from forbidden regions to guiding virtual fixtures, but all of them modify the information between master and slave to improve telepresence and performance. Virtual fixtures are used to assist the operator with the task at hand. They can be used to increase operator performance to expert levels and even beyond. Just like a ruler allows a person to draw a perfectly straight line, virtual fixtures allow operators to perform better while requiring less effort and concentration.

1.4 State-of-the-Art

The purpose of creating a MMS is to have the functionality of a manipulator arm combined with the locomotive capabilities of a mobile platform. MMS are most useful for performing tasks either too difficult or too dangerous for a human to perform. They can be used as assistive devices for people with limited mobility, or for strength by making it possible to lift and move objects many times heavier than a human would be able to. MMS are ideal for situations such as radioactive environments, collapsed buildings, extreme temperature, underwater, or extra-terrestrial environments such as space. MMS can also perform tasks such as bomb diffusal and disposal in place of risking a human life. While it would be easiest to have the MMS complete these tasks on its own, some situations require a human operator. In these cases there are many options for input devices to control MMS, and they can vary in complexity from a single input device, to multiple input devices, to an elaborate software based controller.

1.4.1 Mobile Manipulator User Input Devices

Mobile manipulators designed for real-time operator control require a user input device. MMS can be controlled using spatial joysticks as shown in [12]. More advanced spatial joysticks known as haptic devices have been implemented on MMS [13]. Haptic devices provide force-feedback information from the manipulator allowing the operator to "feel" the forces and torques at the end-effector, providing an improved telepresence. Complex MMS require more than just a pose or trajectory, and therefore have much more intricate input devices. These are often software programs run on computers, or complex hand held devices similar to the way "Andros", "Packbot", and "Talon" are controlled. Images of these robots and their control units are shown in Figures 1.8, 1.9, and 1.10.

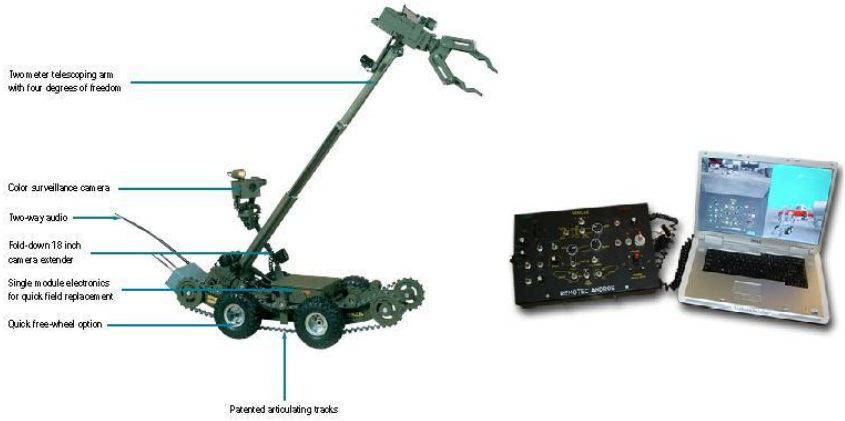


Figure 1.8: Andros MMS with Controller [14]

Figure 1.9: Packbot MMS with Controller [15]

Figure 1.10: Talon MMS with Controller [16]

Controlling a manipulator through the use of a computer allows for teleoperation over much larger distances with the use of the Internet [17]. Through the use of a P300 Brain Computer Interface (BCI), a BCI2000 system was developed and implemented to control a 9-DOF wheelchair-mounted robotic arm system [18]. This system is meant for people with limited use of their lower and upper extremities. The BCI2000 displays an array of Cartesian motions, and by focusing on a certain object within the array the user is able to send commands to the manipulator.

While there are many different types of input devices the simplest of these is a spatial joystick. A spatial joystick is a single input device that only requires one hand for control of the manipulator, giving the operator freedom to perform other tasks simultaneously. Spatial joysticks can be positioned using up to 6-DOF, and therefore can define any 3-dimensional pose. In MMS consisting of a holonomic platform and an articulated manipulator, with the right programming, only a pose or trajectory is required for input and, therefore, a spatial joystick is an ideal choice of user input devices.

1.4.2 Redundancy Resolution

Mobile manipulators are most often designed with a manipulator arm having enough DOF to execute a given task, and is then paired with a mobile platform where it gains an additional 2-3-DOF. With more DOF than is required to specify a pose in the task-space, the system is considered redundant. In a kinematically redundant system a position vector $\mathbf{p}$ (containing both x-y-z position and $\alpha - \beta - \gamma$ orientation as defined in Figure 1.11) can be defined using as many as an infinite number of combinations of joint positions represented by vector $\boldsymbol{\theta}$. Finding the value of $\mathbf{p}$ given the value of $\boldsymbol{\theta}$ employs the use of the forward displacement solution, and the inverse displacement solution yields the value of $\boldsymbol{\theta}$ based on the value of $\mathbf{p}$.

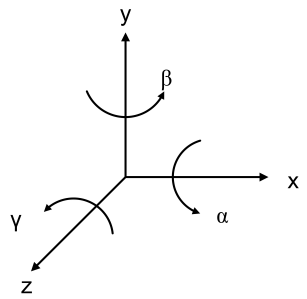


Figure 1.11: 6-DOF Coordinate System

The velocity of the end-effector is defined as $\mathbf{V} = \{ \begin{smallmatrix} v \\ \omega \end{smallmatrix} \}$ (where $\boldsymbol{v}$ is the translational velocity and $\boldsymbol{\omega}$ is the angular velocity) is related to the joint rates $\dot{\boldsymbol{\theta}}$ by the Jacobian matrix $\mathbf{J}$:

$$\mathbf{V} = \mathbf{J}\dot{\boldsymbol{\theta}} \tag{1.1}$$

In an ideal situation the Jacobian is square and invertible. In the case the Jacobian is square and not invertible, i.e., $|\mathbf{J}| \neq 0$ the robot is at a singularity where it loses one or more DOF.

For redundant systems, the Jacobian is not square and therefore never invertible. An alter-

native must be found to resolve the infinity of possible solutions to the inverse displacement and velocity problems.

The following discusses methods of redundancy resolution that have been implemented to date, as well as analyzes the strengths and weaknesses of these techniques. It will focus mainly on instantaneous or local redundancy resolution based on velocity through the use of the manipulator's Jacobian matrix. Global optimization is not ideal since the increased computational requirements make it impractical for a real-time application in which the end-effector is constantly changing trajectory [19].

1.4.3 Kinematic Control of Redundant Manipulators

If $\mathbf{p}$ is an $(m \times 1)$ vector of task variables and $\boldsymbol{\theta}$ is an $(n \times 1)$ vector of joint variables, $\mathbf{p}$ is related to $\boldsymbol{\theta}$ through function $f(\boldsymbol{\theta})$ in the form $\mathbf{p} = f(\boldsymbol{\theta})$, the Jacobian is defined as $\frac{\partial f}{\partial \boldsymbol{\theta}}$, having dimensions $(m \times n)$ and can be seen represented as $\mathbf{J}$ in Equation (1.1). In the case where the dimensions of the Jacobian are $m > n$ the manipulator is redundant and therefore the inverse kinematic solution can yield infinite results. This redundancy can be exploited for avoidance of obstacles and mechanical joint limits, minimization of joint actuator power consumption, or any constraint one wants to add to the system. Redundancy allows for better avoidance of singularities, which occur when the Jacobian matrix $\mathbf{J}$, at a configuration $\boldsymbol{\theta}$, has rank less than m. These singularities cause reduced manipulability resulting in the loss of one or more DOF. Techniques discussed for redundancy resolution can be divided into four categories: Jacobian Based Techniques; Gradient Projection; Augmented Task Space; and Configuration Based Control.

1.4.3.1 Jacobian Based Techniques

Jacobian based techniques introduce a square invertible matrix $\mathbf{K}$ based on the Jacobian matrix $\mathbf{J}$. Following, are a few popular methods for defining $\mathbf{K}$. Whitney [20] used the

Moore-Penrose pseudo-inverse of the Jacobian in the form

$$\mathbf{V} = \mathbf{J}^{\dagger}\dot{\theta} \tag{1.2}$$

and defined $\mathbf{J}^{\dagger} = \mathbf{J}^{\mathbf{T}}(\mathbf{J}\mathbf{J}^{\mathbf{T}})^{-\mathbf{1}}$. Although at first this seems like a suitable solution, joint velocities are only minimized instantaneously and can become very large near singular configurations, thus kinematic singularities are not properly avoided [21]. By using a damped least-square inverse Jacobian $\mathbf{J}^{*} = \mathbf{J}^{\mathbf{T}}(\mathbf{J}\mathbf{J}^{\mathbf{T}} + \lambda^{2}\mathbf{I})^{-1}$, where $\mathbf{I}$ is the identity matrix and $\mathbf{J}^{*}$ is non-singular for the entire workspace of the manipulator, will provide an approximate inverse kinematic solution. The downfall is selecting a suitable damping factor λ weighing the minimum norm solution against the minimum task tracking error [19]. The biggest problem using the Moore-Penrose pseudo-inverse or the damped least-square inverse is repeatability of joint trajectories for a given task trajectory is not preserved [22]. Shamir and Yomdin proposed a mathematical condition on matrix $\mathbf{K}$ to make the solution repeatable [23]. These conditions are not discussed further since any difference between the actual initial joint setting and the calculated one will lead to a loss of repeatability, leaving unrealistic room for error in implementation. If a closed loop equation is desired, $\mathbf{V}$ can be replaced by $\mathbf{V} = \mathbf{V}_{d} + \Lambda\boldsymbol{e}$, $\boldsymbol{e}$ being the error $\boldsymbol{e} = \mathbf{V}_{d} - \mathbf{V}$ where $\mathbf{V}$ is the actual velocity and $\mathbf{V}_{d}$ is the desired velocity calculated from the inverse kinematic solution using $\mathbf{K}$ as the Jacobian and Λ is a positive definite matrix that will shape the error convergence [24]. For a computationally simpler solution, a transpose based solution can be used in the form

$$\mathbf{V} = \mathbf{J}^{\mathbf{T}}\Lambda\boldsymbol{e} \tag{1.3}$$

which eliminates numerical instabilities that occur at kinematic singularities by not requiring a pseudo-inverse of the Jacobian [25].

1.4.3.2 Gradient Projection Method

The gradient projection method uses the following solution:

$$\mathbf{V} = \mathbf{J}^{\dagger}\dot{\boldsymbol{\theta}} + [\mathbf{I} - \mathbf{J}^{\dagger}\mathbf{J}]\dot{\boldsymbol{\theta}}_{\mathbf{0}} \tag{1.4}$$

where $\dot{\boldsymbol{\theta}}_{\mathbf{0}}$ is an arbitrarily chosen joint velocity vector with dimensions $(n \times 1)$. A full rank submatrix of the Jacobian can be used to avoid explicitly calculating $\mathbf{J}^{\dagger}$ for computational efficiency [26]. A common use of the gradient projection method solves redundancy by optimizing for a scalar cost function $h(\boldsymbol{\theta})$, for example using $\dot{\boldsymbol{\theta}}_{\mathbf{0}} = \left(\frac{\partial h}{\partial \boldsymbol{\theta}}\right)^{T}$. Cost functions can be made for the avoidance of mechanical joint limits [27], maximization of kineto-static [28] and dynamic [29] manipulability measures, and maximization of other criteria [30].

1.4.3.3 Augmented Task Space

Sciavicco and Sicillano [31] proposed the use of an augmented task space for redundancy resolution. Doing so involves creating as many additional task constraints for the system as there are degrees of redundancy. The constraints take the form of

$$\boldsymbol{x} = f_x(\boldsymbol{\theta}) \tag{1.5}$$

The forward kinematic solution then takes the form

$$\begin{bmatrix} \boldsymbol{p} \\ \boldsymbol{x} \end{bmatrix} = \begin{bmatrix} f(\boldsymbol{\theta}) \\ f_x(\boldsymbol{\theta}) \end{bmatrix} \tag{1.6}$$

The velocity problem becomes

$$\mathbf{V}_a = \mathbf{J}_{\mathbf{a}}\dot{\theta} \tag{1.7}$$

With the conditions

$$\mathbf{J_a} = \begin{bmatrix} \mathbf{J} \\ \\ \mathbf{J_x} \end{bmatrix}, \mathbf{J_x} = \frac{\partial \boldsymbol{x}}{\partial \boldsymbol{\theta}} \tag{1.8}$$

the inverse kinematic equation becomes

$$\dot{\boldsymbol{\theta}} = \mathbf{K_a}\mathbf{V}_a \tag{1.9}$$

The above is the case Baillieul called the extended Jacobian technique [32]. Note that $\mathbf{K_a} = \mathbf{J_a^{-1}}$. For a closed loop solution, $\mathbf{V}_a$ is replaced by $\mathbf{V}_{ad} + \Lambda_a \boldsymbol{e_a}$ as discussed in Section 1.4.3.1. The augmented task space approach is an attractive one since it is repeatable for any initial joint setting [33].

When the extended or augmented Jacobian becomes singular and the robot is not, an algorithmic singularity occurs. Thus, the problem with this technique is that it is prone to algorithmic singularities [34]. Nakamura, Hanafusa, and Yoshikawa [35] proposed a solution where an algorithmic singularity will not interfere with the whole augmented task, demonstrated in Equation (1.10). Constraints represented by vectors $\mathbf{x}$ and $\mathbf{y}$ are given lower priority than the main task $\mathbf{V}$, and give a solution in the form

$$\dot{\mathbf{p}} = \mathbf{J}^{\dagger}\dot{\boldsymbol{\theta}} + (\mathbf{I} - \mathbf{J}^{\dagger}\mathbf{J})\tilde{\mathbf{J}}_{\mathbf{x}}^{\dagger}(\dot{\mathbf{x}} - \mathbf{J_x}\mathbf{J}^{\dagger}\dot{\mathbf{V}}) + (\mathbf{I} - \mathbf{J}^{\dagger}\mathbf{J})(\mathbf{I} - \tilde{\mathbf{J}}_{\mathbf{x}}^{\dagger}\tilde{\mathbf{J}}_{\mathbf{x}})\mathbf{y} \tag{1.10}$$

where

$$\tilde{\mathbf{J}}_{\mathbf{x}} = \mathbf{J_x}(\mathbf{I} - \mathbf{J}^{\dagger}\mathbf{J}) \tag{1.11}$$

While Equation (1.10) is acceptable, it comes at a very high computational cost like other methods discussed in Sections 1.4.3.1 to 1.4.3.3.

1.4.3.4 Configuration Based Control

For some mobile-manipulator systems, the redundancy is controlled by locking one or more joints to lower the DOF of the system [20]. This is done for the Canadarm2 and Dextre for the International Space Station [12]. Although this negates any benefits of redundancy, the inverse kinematic problem can be solved easily. If this technique is only used when the robot is in certain configurations, configuration based control emerges. With configuration based control, one is able to separate the manipulator from its mobile base at configurations that are singular to the manipulator. Takubo et. al. [36] do so through the use of a virtual impedance wall. In many cases the manipulator arm's movement is quicker, more accurate, and stable than that of its base, and consumes much less power. The virtual impedance wall is designed to maximize manipulability and stability at any time, while minimizing power consumption [36]. The virtual impedance wall is a workspace created to maximize the manipulability of the robot and keep the center of gravity close enough to the base so that it does not tip over. When the end-effector reaches the limit of this preferred operating area, the mobile base moves away from the impedance wall with a repulsive force simulated by a spring and damper system. Shown in Figure 1.12 is a visual representation of this method.

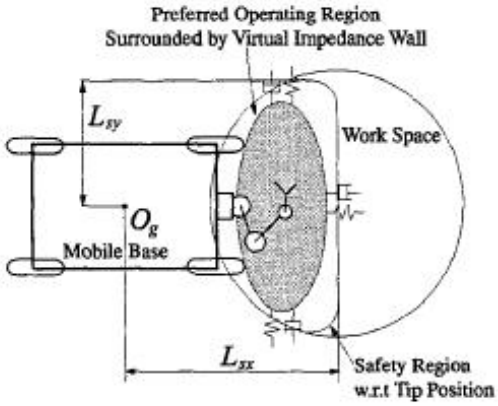


Figure 1.12: Preferred Operating Region Design [36]

1.4.4 Haptic Control of a Mobile Manipulator

Figure 1.13: Novint “Falcon” [37]

Figure 1.14: Haption’s “Virtuose” [38]

Figure 1.15: Sensable Technologies “Phantom” [39]

In recent years haptic devices have shown a decrease in cost and an increase in applications [40]. There are many manufacturers that make haptic devices such as Novint’s “Falcon” (Fig. 1.13), Haption’s “Virtuose” (Fig. 1.14), and Sensable Technologies “Phantom

Omni" (Fig. 1.15) but the great importance lies in how they are used. Haptic devices are not limited to the control of mobile manipulators, but can be used in classrooms, hospitals, for visually impaired, military and flight training, art, and entertainment [41–47].

When selecting a haptic device one must consider the following criteria, common in all haptic devices [48]:

Degrees of Freedom (DOF): Typically the DOF of the device matches the DOF of the application

Workspace: The total usable area of the device

Position Resolution: How accurate the device is

Continuous Force: The maximum force the device can exert for an extended period of time

Maximum Force/Torque: The maximum instantaneous force the device can produce

Maximum Stiffness: This refers to the stiffness of a virtual fixture, and is dependent on a number of other physical properties

System Latency: How quickly the system can respond to a change in the input

Haptic Update Rate: The frequency the device can change the haptic forces

Inertia: The perceived mass of the devices end-effector

As well as the physical characteristics, one must consider the functionality of the Application Programming Interface (API). The API allows the programmer to incorporate the haptic device into the software platform. A robust API is desirable when the haptic device is being used for a variety of tasks, but when the device is selected for a specific task the programmer must ensure the API is capable of performing the required operation.

There have been many experiments testing the effectiveness of haptic devices as training tools in a variety of applications. These devices improve training time in simple tasks such as target-hitting [49] and have been used to assist in tasks as complex as microrobotic intracellular injection [50]. Haptics have been shown to decrease surgery time in stitching applications [51] as well as improve more difficult procedures such as palpatory techniques [52]. Haptic devices can improve a sense of "togetherness" in collaborative tasks [53], even when latency is involved [54]. Haptics have also been used in place of controlling individual DOF of large hydraulic manipulators to decrease training time on the manipulator [55]. Haptic devices are often used to control mobile manipulators, and through simulation and experimental study, they have been shown to improve the quality of mobile robot teleoperation [56].

Just like a computer with no operating system, a haptic device's usefulness is significantly reduced without the use of virtual fixtures. Virtual fixtures are most often designed with a task in mind, and therefore the types and styles of virtual fixtures vary as much as the types of tasks one can perform with a haptic device. There is an overwhelming number of published papers showing how virtual fixtures improve teleoperation in simple tasks like pick-and-place [11] to robotic cardiac surgery [57]. Virtual fixtures can increase performance in manipulation tasks, especially when external forces are present. In micromanipulation tasks, performance can be improved by 52% through using virtual fixtures to prevent the influence of microphysics on path planning and handling tasks [58]. In admittance devices, compliance can significantly affect performance but use of virtual fixtures can greatly reduce this effect [59].

1.5 Problem Statement

Many of the methods of redundancy resolution discussed in Section 1.4 are best utilized when the desired trajectory of the end-effector is already known. While they do provide valid solutions for the inverse kinematic problem, it is too computationally intensive for real-time control. Many of these methods have been designed for single systems with multiple degrees of redundancy and predefined trajectories but do not take the particular case of a teleoperated MMS into consideration. However, there are methods of redundancy resolution designed specifically for mobile-manipulator systems. Using a configuration based command strategy that separates control of the manipulator arm from its mobile base makes calculating the inverse kinematic solution simple and therefore computationally efficient, which is important for real-time control. By modeling the MMS as two subsystems, one is able to assign priority to one of the subsystems and optimize the entire system for certain criteria. Minimizing the mobile base movement will optimize power consumption, stability, and manipulability. For that reason a virtual impedance wall or similar control algorithm would be an ideal choice for real-time teleoperated control of a MMS using a single spatial joystick. Although freeing up a hand for other activities and introducing a more intuitive command strategy will improve the teleoperation experience, simplicity alone is not sufficient. Virtual fixtures will be used to increase telepresence and task accuracy as well as efficiency.

1.6 Goals

The successful outcome of this work will depend on three main goals. First, a suitable test-bed is required to test command strategies. The second goal is to use haptic technology to create a simple and intuitive command strategy for teleoperating a MMS. Finally, virtual fixtures will be used to increase performance and telepresence.

1.6.1 Test-Bed MMS

In many cases in the literature, command strategies are implemented in simulation, however, this work will be done on a real MMS. Simulations are excellent for implementing and testing new command strategies in a variety of situations quickly but one may not be able to expect the same performance from an operator in a real world scenario. Using an actual MMS, command strategies can be tested in a workplace environment and can provide a more accurate representation of how the operator will perform. A MMS will be constructed to serve as a test-bed for testing and verification of the remaining two goals.

1.6.2 Haptic Command Strategy

This research will also include the use of haptic technology. A haptic device capable of 3-dimensional positioning will be used to control the MMS. Teleoperation must require only a single hand, and must occur in real-time. A command strategy will be developed for use with the haptic device and MMS. The command strategy must be simple to learn and intuitive while operating.

1.6.3 Virtual Fixtures

Virtual fixtures will be an integral part of the command strategy. Improved telepresence will be achieved by using virtual fixtures to transmit critical information about the MMS to the operator. Virtual fixtures will assist the operator in performing the required tasks and preventing undesirable actions such as collisions.

1.7 Contributions

The contributions of this thesis are:

1. Development of a MMS test-bed using a 3-DOF manipulator and omnidirectional mobile base.

2. Design and implementation of a novel command strategy for teleoperating MMS using a single haptic input device. The command strategy uses virtual fixtures and three states of operation: near-target manipulation, off-target manipulation, and transportation modes.

1.8 Organization

The remainder of this thesis is organized as follows. Chapter 2 presents the hardware used for the new command strategy. This includes the MMS, haptic device, and peripheral hardware. The software systems that are used in this system are also discussed. Chapter 3 discusses preliminary command strategies that were tested on the MMS, and introduces the new command strategy proposed in this work. Chapter 4 explains the test set-up used to test the new command strategy, and presents the results of testing. Chapter 5 makes conclusions based on the results from the previous chapter and suggests direction for future work.

Chapter 2

Mobile-Manipulator System

This chapter presents an overview of the Mobile-Manipulator System (MMS) used in this work, namely the Omnibot MMS shown in Figure 2.1. As with any MMS, the Omnibot MMS consists of two defining components, the mobile base and manipulator arm. The omnidirectional mobile base called the Omnibot and a manipulator made from Powercube modules form the Omnibot MMS. To teleoperate, the "Falcon" haptic joystick is used. As well, a number of computers are used for communication between the operator and the hardware systems.

2.1 Manipulator

The 3-DOF (degree-of-freedom) manipulator shown in Figure 2.2 is capable of position control but not orientation. The user-defined safe operating space is a rectangle that fits inside the manipulator's workspace and therefore the end-effector can be positioned in any Cartesian coordinate within it. Motion is achieved through the use of three SCHUNK rotary modules called Powercubes.

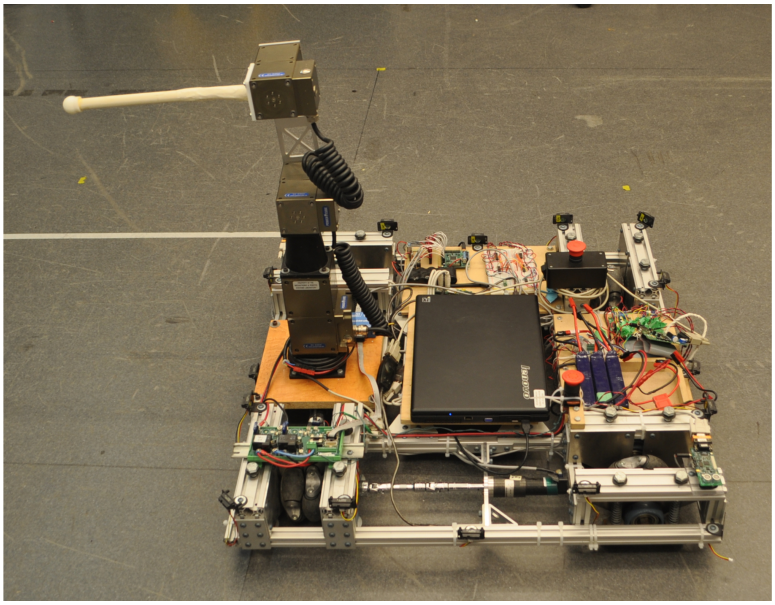


Figure 2.1: The Omnibot Mobile-Manipulator System

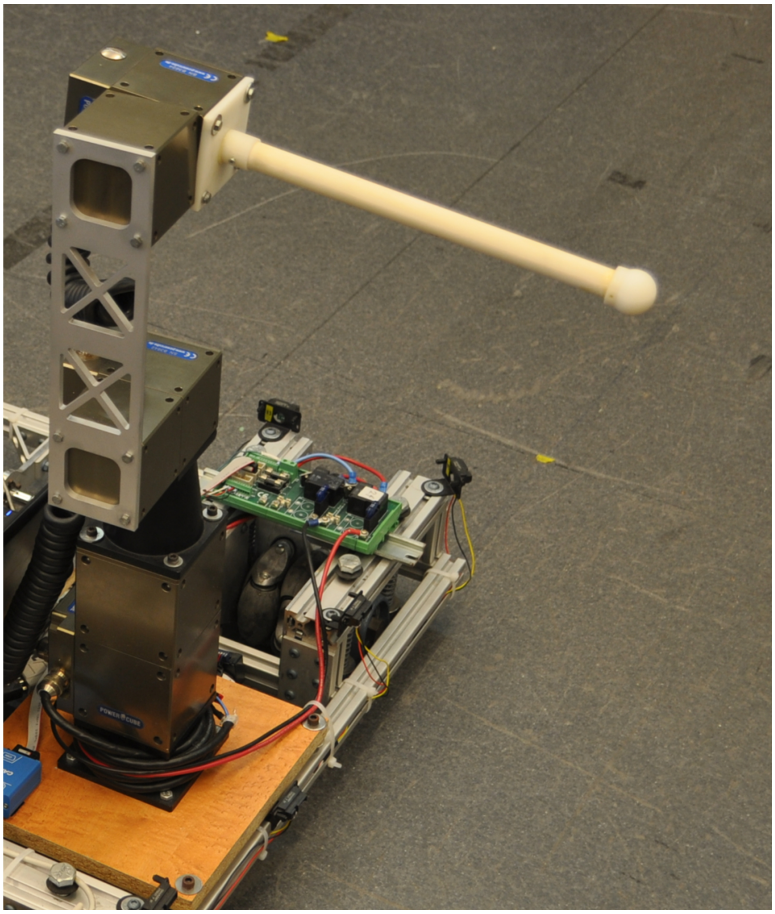
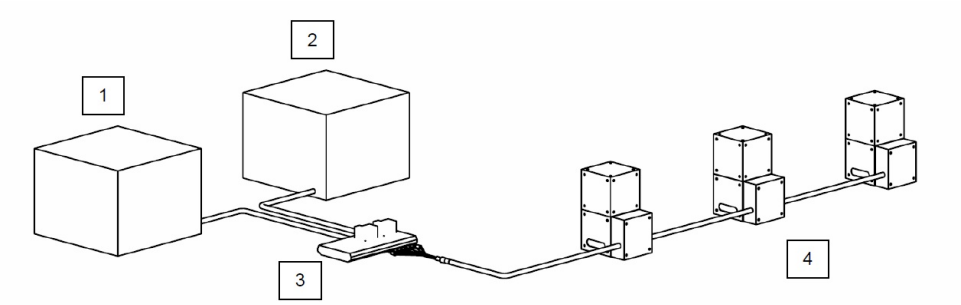


Figure 2.2: The Omnibot MMS Manipulator Arm



1 Power supply (transformer with rectifier or switching power pack)
2 PC, PLC or other suitable control
3 PowerCube Terminal Block
4 PowerCube modules 1 to n

Figure 2.3: Powercube Circuit Diagram for Electrical Connection [60]

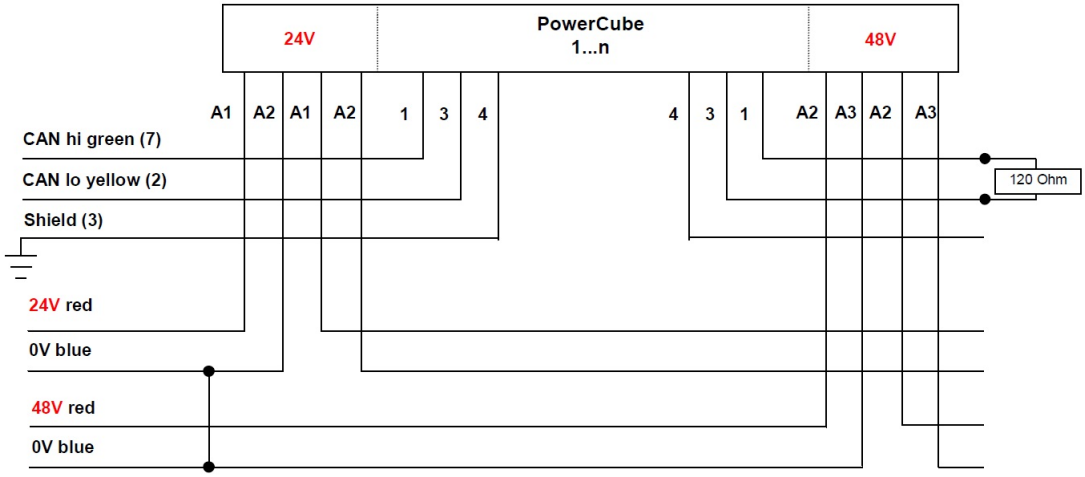


Figure 2.4: The Schematic Layout for a CAN Bus Using Powercubes [60]

2.1.1 Components

The Powercube arm operates using three main components. Most importantly are the individual rotary modules that provide the arm's mobility. Each rotary module is called a "Powercube" and two sizes of Powercubes are used in the manipulator. The larger of the two is used as the base link for the arm. It has a maximum velocity of 2.6 rad/s, maximum acceleration of 10 rad/s^2, and can handle a maximum of 30 A of current. Two smaller rotary modules complete the last two links of the manipulator. The joints are arranged in a $R \perp R \parallel R$ layout, where R denotes a revolute joint, $\perp$ denotes two joints perpendicular to

one another, and || denotes two joints parallel to one another. Similar to the larger module, the smaller modules use a power supply of 24 V DC with the same maximum velocity and acceleration as the larger module, but current capability is limited to 15 A. Full technical details of the modules can be found in [60] and [61]. When in motion, the arm consumes between 1 - 1.5 A of current. When not in use, electronic brakes are applied to the modules to keep them from moving. The brakes consume 0.3 - 0.4 A of current. The power supply on-board the Omnibot MMS has a 5 Ah capacity allowing the manipulator to run 3-5 hours continuously under maximum operating conditions. Additional batteries can be easily added to increase operation time.

The modules are controlled and communicate using a CAN bus. The electrical layout is shown in Figure 2.3. RS232 and Profibus DP communication protocols are available as well, but CAN was selected because commands can be sent to individual modules connected to the bus, or to all at once. A schematic of the CAN bus is shown in Figure 2.4. The CAN bus also allows the controller to send separate commands to the modules and execute them simultaneously. A USB-CAN device on the Omnibot MMS allows commands and information to be sent and received between the computer system and Powercube modules. The terminal block provides the required end-point for the CAN bus and an emergency-stop button for the manipulator. Fuses on the terminal block ensure that the modules will not get damaged if the current exceeds safety limits. Figure 2.5 is an image of the terminal block and USB-CAN interface.

2.1.2 Software Control

The Powercube module controller is written in C programming language. The Application Programming Interface (API) provided by SCHUNK allows the programmer to use a library of commands to control the Powercube modules. A short list of the most useful

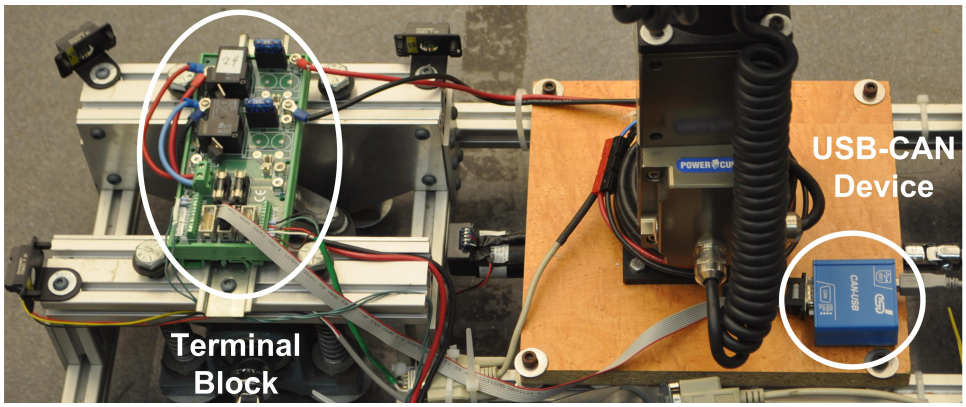


Figure 2.5: Powercube manipulator interface devices

commands are as follows:

PCube_openDevice() Opens the specified Powercube device

PCube_resetAll() Resets all modules connected to the bus

PCube_moveRamp() Moves a module to a specified position with a specified maximum velocity and acceleration

PCube_moveVel() Moves a module at a specified velocity

PCube_getPos() Returns the position of the specified module

The full documentation on the Powercube API can be found in [61].

2.1.3 Kinematics

Since the Powercube manipulator was built specifically for this work, there was no kinematic model available. In this section, the manipulator kinematic model is developed.

2.1.3.1 Denavit-Hartenberg Parameters

Using Craig's modified Denavit-Hartenberg (D-H) Parameters [62], the D-H table in Table 2.1 describes each link's position and orientation relative to the previous link. All angles are in radians and lengths in meters. Since link lengths a and b can be easily changed, their actual values are not used throughout the calculations to enable future modifications. For the current configuration of the manipulator the values of a and b are 0.228 m and 0.3125 m, respectively.

Table 2.1: Denavit-Hartenberg Parameters

F_{i-1}	α_{i-1}	a_{i-1}	d_i	θ_i	F
0	0	0	0	θ_1	1
1	$-\frac{\pi}{2}$	0	0	θ_2	2
2	0	a	0	θ_3	3
3	$\frac{\pi}{2}$	0	b	0	ee

2.1.3.2 Homogeneous Transformation Matrix

The homogeneous transformation matrix is derived from the D-H parameters in Table 2.1 and is used to describe the end-effector's frame of reference relative to the base frame of reference. In order to define the pose (position and orientation) of the manipulator the homogeneous transformation matrix is required. A point in the manipulator's "world" frame (the frame of reference to which its base link is attached to) can be transformed to a point in the end-effector's frame, using:

$$
{}^{0}_{ee}\mathbf{T} = \begin{bmatrix} c_1c_{23} & -s_1 & c_1s_{23} & c_1c_2(s_3b+a)+c_1s_2c_3b \\ s_1c_{23} & c_1 & s_1s_{23} & s_1c_2(s_3b+a)+s_1s_2c_3b \\ -s_{23} & 0 & c_{23} & -s_2(s_3b+a)+c_2c_3b \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.1}
$$

where c_{ij} and s_{ij} denote $\cos(\theta_i + \theta_j)$ and $\sin(\theta_i + \theta_j)$ respectively.

In the case of the Omnibot MMS, the manipulator's world frame is the local frame of the Omnibot.

To define the pose of the entire MMS, the homogeneous transformation matrix of the Omnibot is required as well. For the Omnibot's homogeneous transformation matrix ${}^{W}_{Omni}\mathbf{T}$, the assumption that the Omnibot is driving exclusively on flat ground is made and therefore all z elements (height) is set to 0. The x and y of ${}^{W}_{Omni}\mathbf{T}$ refer to the position of the Omnibot in the room, and θ is the orientation:

$$ {}^{W}_{Omni}\mathbf{T} = \begin{bmatrix} c\theta & -s\theta & 0 & x \\ s\theta & c\theta & 0 & y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.2} $$

2.1.3.3 Forward Kinematic Solution

Derived from the homogeneous transformation matrix, the manipulator's forward kinematic solution calculates the end-effector location (relative to the base link) from each link's position and is found as:

$$ x = c_1 s_{23} b + c_1 c_2 a \tag{2.3} $$

$$ y = s_1 s_{23} b + s_1 c_2 a \tag{2.4} $$

$$ z = c_{23} b - s_2 a \tag{2.5} $$

2.1.3.4 Inverse Kinematic Solution

The inverse kinematic solution is found by solving Equations (2.3) to (2.5) for θ_1, θ_2, and θ_3, yielding:

$$\theta_1 = Atan2\Big(y(s_{23}b + c_2a), x(s_{23}b + c_2a)\Big) \tag{2.6}$$

$$\theta_2 = Atan2(-s_3b - a, c_3bc_2) \pm Atan2\Big(\sqrt{(c_3bc_2)^2 + (-s_3b - a)^2 - z^2}, z\Big) \tag{2.7}$$

$$\theta_3 = Atan2\left(\frac{x^2 + y^2 + z^2 - a^2 - b^2}{2ab}, \pm\sqrt{1 - \left(\frac{x^2 + y^2 + z^2 - a^2 - b^2}{2ab}\right)}\right) \tag{2.8}$$

where $Atan2$ denotes a quadrant corrected arc-tangent function.

Using the inverse kinematic solution, the required joint angles can be calculated for a given end-effector position.

2.1.3.5 Jacobian Matrix

To calculate the end-effector's velocity based on the joint rates, the Jacobian matrix is used. The velocity solution for the manipulator is given by:

$$\mathbf{V_{arm}} = \mathbf{J_{arm}}\dot{\boldsymbol{\theta}} \tag{2.9}$$

$$\begin{Bmatrix} v_x \\ v_y \\ v_z \end{Bmatrix} = \begin{bmatrix} -s_1s_{23}b - s_1c_2a & c_1c_{23}b - c_1s_2a & c_1c_{23} \\ c_1s_{23}b + c_1c_2a & s_1c_{23} - s_1s_2a & s_1c_{23}b \\ 0 & -s_{23}b - c_2a & -s_{23}b \end{bmatrix} \begin{Bmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \\ \dot{\theta}_3 \end{Bmatrix} \tag{2.10}$$

where $\dot{\boldsymbol{\theta}}$ is the vector of joint velocities, $\mathbf{V_{arm}}$ is the velocity of the end-effector, and $\mathbf{J_{arm}}$

is the Jacobian that maps the end-effector velocity to the joint velocities.

2.1.3.6 Velocity Equations

To find the required joint velocities $\dot{\boldsymbol{\theta}}$ for a given end-effector velocity $\mathbf{V}$, the inverse Jacobian is used $\dot{\boldsymbol{\theta}} = \mathbf{J}_{\mathbf{arm}}^{-\mathbf{1}}\mathbf{V}$ to find $\dot{\theta}_1$, $\dot{\theta}_2$, and $\dot{\theta}_3$.

$$\dot{\boldsymbol{\theta}} = \begin{Bmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \\ \dot{\theta}_3 \end{Bmatrix} = \mathbf{J}_{\mathbf{arm}}^{-\mathbf{1}}\mathbf{V} \tag{2.11}$$

where:

$$\begin{aligned} \dot{\theta}_1 = & \frac{-s_1}{c_2c_1^2a + c_2s_1^2a + s_{23}bc_1^2 + s_1^2s_{23}b}V_x \\ & + \frac{c_1}{c_2c_1^2a + c_2s_1^2a + s_{23}bc_1^2 + s_1^2s_{23}b}V_y \end{aligned} \tag{2.12}$$

$$\begin{aligned} \dot{\theta}_2 = & \frac{(-c_1s_{23}b - c_1c_2a + s_1a)s_{23}}{a(s_{23}^2bc_1^2s_2 + s_{23}bc_1^2c_2c_{23} + s_{23}c_1^2c_2as_2 + c_1^2c_{23}c_2^2 + s_{23}^2bs_1^2s_2 + s_{23}bs_1^2c_{23}c_2 + s_1^2c_2as_{23}s_2 + s_1^2c_2^2ac_{23})}V_x \\ & - \frac{(s_{23}s_1b + s_1c_2a + c_1a)s_{23}}{a(s_{23}^2bc_1^2s_2 + s_{23}bc_1^2c_2c_{23} + s_{23}c_1^2c_2as_2 + c_1^2c_{23}c_2^2a + s_{23}^2bs_1^2s_2 + s_{23}bs_1^2c_{23}c_2 + s_1^2c_2as_{23}s_2 + s_1^2c_2^2ac_{23})}V_y \\ & - \frac{c_{23}}{(s_{23}s_2 + c_{23}c_2)a}V_z \end{aligned} \tag{2.13}$$

$$\begin{aligned} \dot{\theta}_3 = & -\frac{-c_1s_{23}b - c_1c_2a + s_1a}{(c_2c_1^2c_{23} + s_1^2c_{23}c_2 + s_{23}c_1^2s_2 + s_1^2s_{23}s_2)ba)}V_x \\ & + \frac{s_{23}s_1b + s_1c_2a + c_1a}{(c_2c_1^2c_{23} + s_1^2c_{23}c_2 + s_{23}c_1^2s_2 + s_1^2s_{23}s_2)ba}V_y \\ & - \frac{s_2a - c_{23}b}{(s_{23}s_2 + c_{23}c_2)ba}V_z \end{aligned} \tag{2.14}$$

The inverse velocity solution for the base is given by:

$$\dot{\mathbf{q}} = \mathbf{J_{base}} \mathbf{V_{base}} \tag{2.15}$$

$$\begin{Bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \dot{q}_3 \\ \dot{q}_4 \end{Bmatrix} = \begin{bmatrix} -\sin\theta & \cos\theta & l\cos(45^\circ) \\ -\cos\theta & -\sin\theta & l\cos(45^\circ) \\ \sin\theta & -\cos\theta & l\cos(45^\circ) \\ \cos\theta & \sin\theta & l\cos(45^\circ) \end{bmatrix} \begin{Bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{Bmatrix} \tag{2.16}$$

where $\dot{\mathbf{q}}$ represents the velocities of each wheel, θ represents the orientation of the Omnibot, l is the distance from the geometric center of the Omnibot to the wheels, and $\mathbf{V_{base}}$ represents the desired translational and rotational velocities.

2.2 Omnidirectional Base

The mobile base of the MMS called the Omnibot [63] can be seen in Figure 2.6. It is a holonomic (also known as omnidirectional) robot which means it has the ability to translate and rotate simultaneously. The Omnibot's motion capabilities make it an ideal base for the MMS because the operator can focus exclusively on the motion of the Omnibot while rotation occurs either autonomously or manually. Holonomic motion is achieved through the use of four orthogonally placed "omni-wheels". A layout of the omni-wheels on the Omnibot is shown in Figure 2.7.

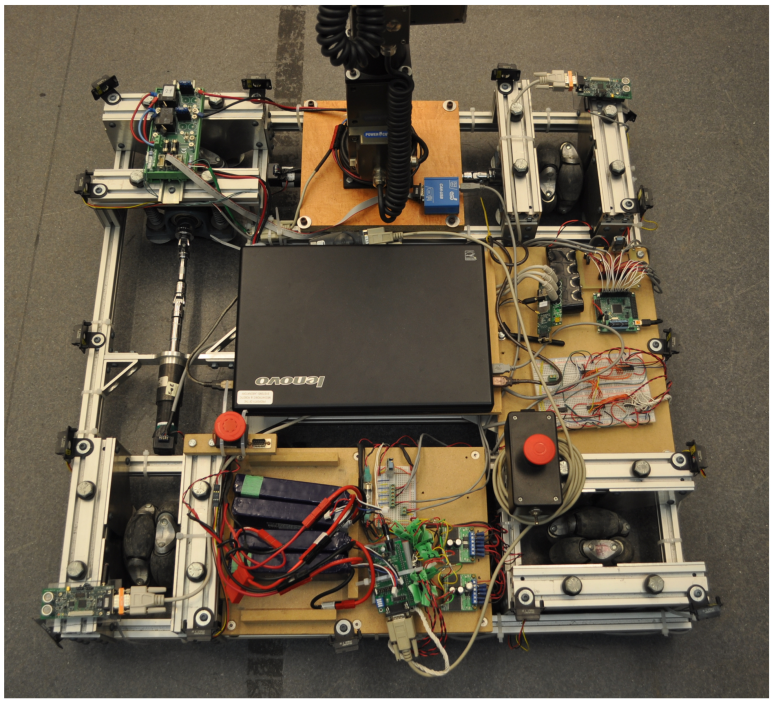


Figure 2.6: The Omnibot Mobile Base

2.2.1 Functional Overview

For locomotion, the Omnibot has four spring dampened omni-wheels that allow it to move omnidirectionally. Each wheel is driven by a MicroMo DC motor, with two Victor 884 motor drivers to run the motors at 12 V using Pulse Width Modulation (PWM) to control the speed. Attached to each motor is an encoder which allows the Omnibot to record odometry through the encoder integrated circuits shown in Figure 2.11. The motor with its encoder is shown in Figure 2.8. A Motorola HCS12 microcontroller performs the kinematic calculations for motion, as well as applies PID (Proportional Integral and Derivative) control to each wheel. The motor driver and HCS12 microcontroller is found in Figure 2.9. Two Crickets (see Section 2.2.2) are mounted at opposing corners of the Omnibot, and communicate with an on-board Gumstix computer (shown in Figure 2.10). Through serial communication, the Gumstix computer sends distance estimates from the Crickets to the on-board laptop computer.

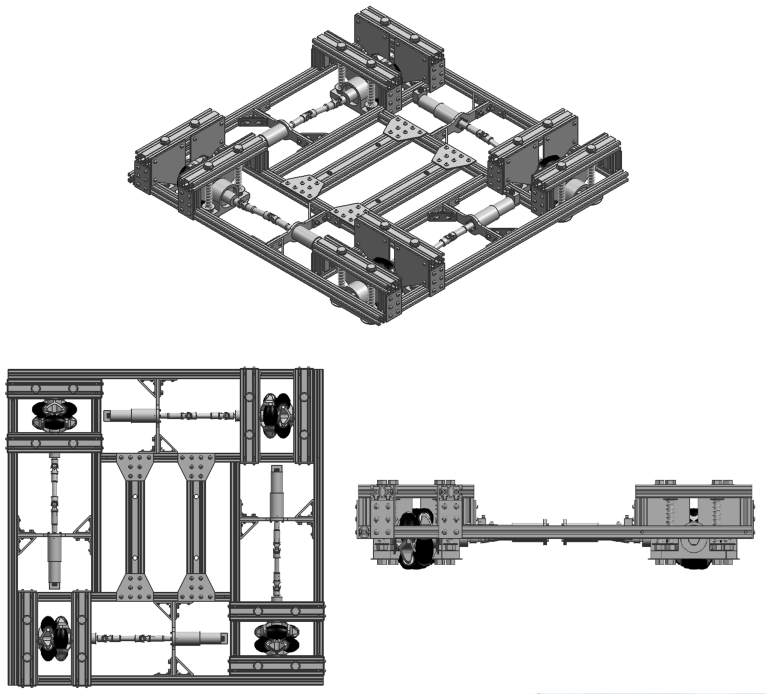


Figure 2.7: The Omnibot's Frame and Omni-wheel layout [63]

The laptop computer uses a Linux operating system and runs Robot Operating System (ROS). More information about ROS can be found at the website: www.ros.org. Using ROS, individual programs called nodes communicate by publishing and subscribing to topics. The laptop performs localization calculations, sends motion commands to the HCS12, controls the manipulator, and receives commands from the joystick remotely. Twelve infrared (IR) sensors mounted around the Omnibot along with an AVR microcontroller allow it to estimate its distance from objects. The microcontroller and one of the IR sensors can be see in Figure 2.11. An emergency stop button is located on the Omnibot (right side on Figure 2.6) and cuts all power to the device when pressed.

2.2.2 Localization

The Omnibot uses a modified Cricket system for localization [64]. Cricket nodes like the one shown in Figure 2.10 are configured either as beacons or listeners. The two Crickets

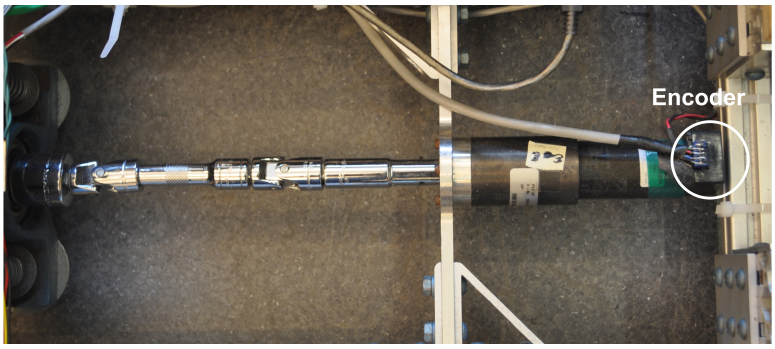


Figure 2.8: The Omnibot's Motor and Encoder

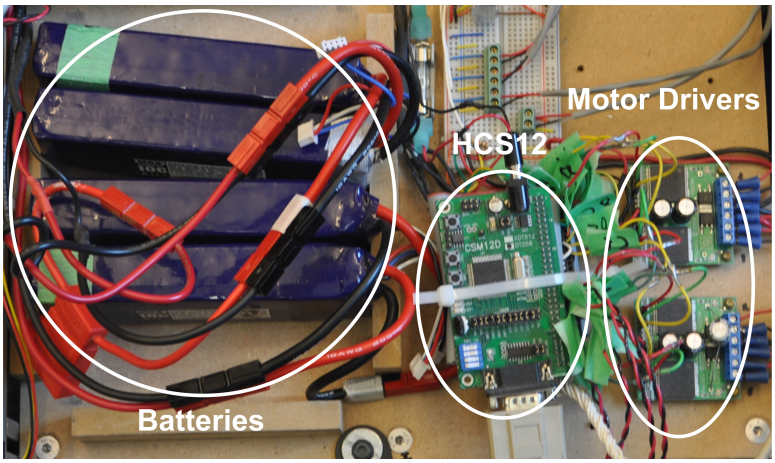


Figure 2.9: The Omnibot's HCS12 Microcontroller and Motor Drivers

located at opposite corners of the Omnibot are set as beacons, and an array of Crickets at fixed locations within the workspace are set as listeners. The system operates using time of flight data similar to the Global Positioning System (GPS). A radio frequency ping is sent out with the beacon's specific information along with and ultrasonic signal. The difference in time of flight of the two signals allows the system to estimate the distance between the beacon and the listener. Using multiple listeners to attain a minimum of three distance estimates, the position of the beacon can be calculated using trilateration. A schematic of the localization setup is shown in Figure 2.12. Using the position estimates of the two beacon nodes the location of the Omnibot's geometric center along with its orientation is calculated. The encoder data from the wheels is used when combining odometry data with the cricket localization to gain a more accurate estimate of the Omnibot's pose. Shown in

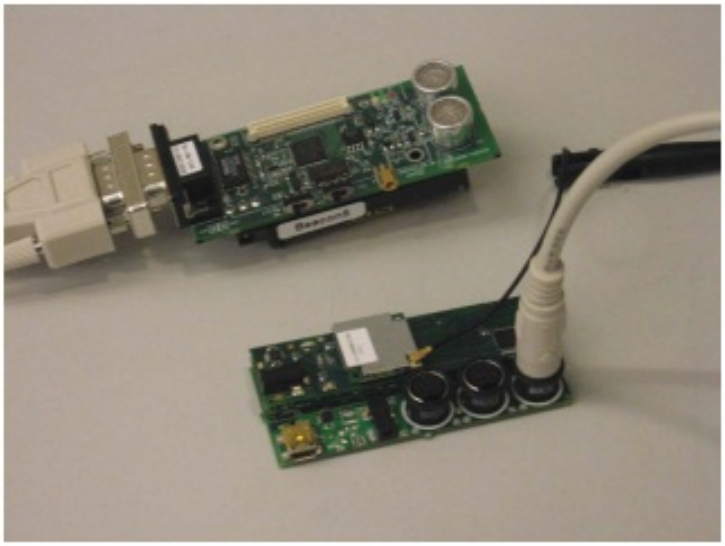


Figure 2.10: Cricket Node (top) and Gumstix Computer (bottom)

Figure 2.11 is the hardware used in the localization system of the Omnibot.

2.2.3 Current Control Methods

At the start of this research, there were two basic control methods implemented on the Omnibot. The operator could either teleoperate the Omnibot, or allow it to navigate autonomously. Under autonomous navigation, the Omnibot will follow preprogrammed waypoints. When teleoperated, the operator controls the Omnibot using a joystick.

2.2.3.1 Autonomous Control

The Omnibot can autonomously follow waypoints that are preprogrammed in the system via a waypoints file. When defining a waypoint, both a position and orientation are given and the Omnibot travels at a fixed speed and rotation, rotating on the spot if it reaches its destination before achieving its target orientation. The translation and rotation velocity are specified for each waypoint individually. The Omnibot is not able to navigate around or

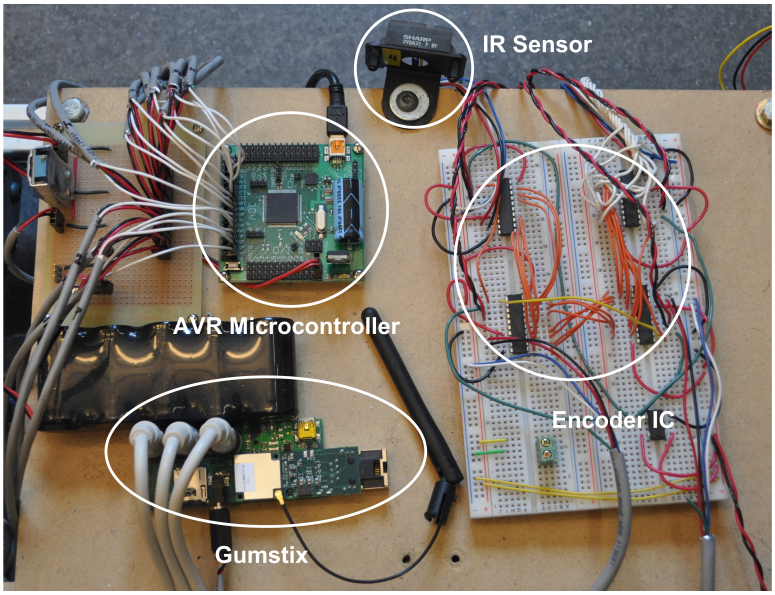


Figure 2.11: Gumstix computer, Encoders and AVR microcontroller

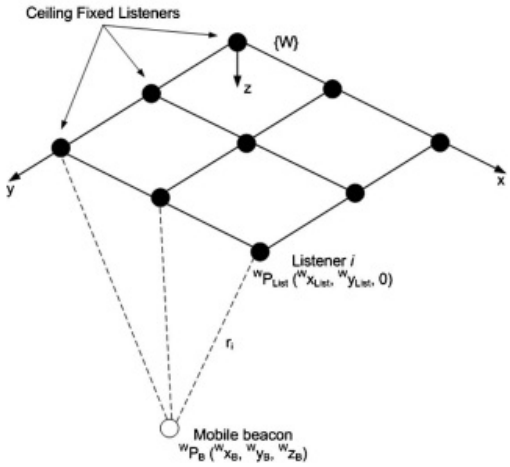


Figure 2.12: Trilateration of a Cricket Mobile Beacon Node [64]

Figure 2.13: 3-DOF Omnibot Joystick

avoid obstacles, because it travels directly to each waypoint in sequence. The operator can also maneuver the Omnibot semi-autonomously by giving it individual waypoints one at a time, and waiting for the Omnibot to reach its destination before giving it the next waypoint. Under semi-autonomous navigation, the translation and rotation velocity can either be specified or kept the same as the previous waypoint.

2.2.3.2 Teleoperation

The operator can teleoperate the Omnibot using a 3-DOF joystick (see Figure 2.13). The joystick can be tilted forward, backward, left, and right to move the Omnibot. The operator can also twist the joystick to control the rotation of the Omnibot. The joystick can be used by connecting it directly to the robot or through wireless communication. When using this joystick to control the Omnibot, the operator controls it in a local frame of reference. When the joystick is pushed forward, the Omnibot moves in whatever direction its front is facing. A "dead-man" switch is used, where the button must always be held down in order for the Omnibot to move. Whenever the button is released, the Omnibot stops.

Direct Control When using direct control, the operator connects the joystick directly to the HCS12 microcontroller on the Omnibot. When the microcontroller receives an analog signal from the joystick, it converts it to a digital signal and sets the desired velocity of each motor using the kinematic model and PID control. A sufficiently long serial cable is required to operate the Omnibot at any significant distance.

Wireless Control When teleoperating the Omnibot without a direct connection, a teleoperation station is used. This station consists of a computer, HCS12 microcontroller, and the 3-DOF joystick connected to it. The microcontroller converts the analog signals of the joystick to digital values, which are received by the computer via a serial connection. A ROS node running on the computer converts the position of the joystick to a motion command that is published to a ROS topic. The motion command is received by a node running on the Omnibot computer through its subscription to the same topic. The node then sends the desired velocity commands to the Omnibot HCS12 via a serial connection and the HCS12 sets the desired motor velocities in the same way it does under direct control.

2.3 Haptic Joystick

The joystick used in previous work on the Omnibot was a 3-DOF device. It had two translational and one rotational DOF. For this research, the Omnibot was fitted with a 3-DOF manipulator having a vertical DOF in place of the Omnibot's rotational DOF. For that reason, the previous joystick was not suitable for teleoperation of the Omnibot's manipulator. A spatial joystick, one that moves in x, y, and z directions is required. In addition, the ability to provide haptic feedback is key in the development of new control methodologies.

Figure 2.14: The Novint Falcon Haptic Joystick

2.3.1 Haptic Devices

A haptic device provides tactile feedback to the operator in the form of forces, vibrations, and/or motions. Using these forces, virtual objects can be created in a computer simulation that a person can interact with as if it was a real object. The sample rate of a haptic device is very important when attempting to create the illusion of virtual reality. For example, a person can easily tell the difference between a 13 kHz and 14 kHz tone, however any video having a frame rate greater than 24 Hz will appear as smooth motion. In order to create the sensation of a smooth surface, a sampling rate between 500 Hz and 1,000 Hz is required.

Haptic joysticks are two-way information devices, where the operator sends position or force information through the joystick, and receives haptic information via the same device.

2.3.2 Novint Falcon

The haptic joystick called the Falcon shown in Figure 2.14 made by Novint Technologies Incorporated, was selected for this work. It is a 3-DOF impedance device and has four buttons located on its end-effector. Using USB 2.0 connectivity, the device can communicate

with Windows operating system based computers. The Falcon has a haptic workspace of 4" $\times$ 4" $\times$ 4" (101.6 mm $\times$ 101.6 mm $\times$ 101.6 mm), with a position resolution of 400 dpi. Due to its parallel structure it cannot create the same maximum force across its workspace, but guarantees a minimum of 8.9 N of force at any point.

There is a coloured LED badge that indicates the calibration status of the device, red for uncalibrated and blue for calibrated. Calibration is as simple as extending the device fully outward, followed by retracting it all the way inward and is required every time the computer it is connected to powers on.

2.3.3 Programming Haptic Forces

Haptic forces created in a computer simulation are sent to the Falcon with the use of Novint's Haptic Device Abstraction Layer (HDAL). The HDAL is designed to be a uniform interface for all of Novint's haptic devices. The software development kit allows the user to access the HDAL through the use of an API. Similar to the API used for the Powercube modules, the Falcon API provides a number of programming functions to the programmer for interfacing with the device. The main functions used from the API are:

hdlToolPosition() Reads the current position of the device

hdlToolButton() Reads the current status of the buttons

hdlSetToolForce() Sets the forces on the device

hdlCreateServoOp() Creates the servo callback function

The servo callback function operates at a frequency of 1,000 Hz. It is this function that allows the programmer to create the illusion of a virtual object. Each time the callback

function executes, the desired forces are calculated and set for the device, and the tool's position and button status is read. The callback function allows the Falcon to run at the required sampling rate for haptic virtual reality, while the rest of the simulation can run at its own pace, allowing the haptics to run in real-time.

2.4 Computers

To complete the teleoperation of the Omnibot MMS, two main computers are required. While a separate computer is used to initialize the localization, its use ends with that function and is not considered a crucial part of the system. As well, the system that synchronizes the Crickets, i.e., the Gumstix, is in fact a computer, but acts more as a microcontroller and is considered one of the Omnibot's embedded controllers. The two main computers used in the teleoperation of the MMS consist of a remote workstation and a laptop mounted onboard the Omnibot.

2.4.1 Remote Workstation

The remote workstation is a Windows computer that the haptic device is connected to. This computer runs a C++ program that communicates to the Omnibot via IP socket communication. This protocol allows teleoperation anywhere both computers can connect to the same network. Using IP communication the system can be modified to operate over the Internet, allowing teleoperation at any distance, although timing delays become an issue.

The remote workstation receives pose data from the Omnibot computer and transmits the position and button status of the Falcon. The pose data received contains the x, y and orientation (θ) of the Omnibot and the x, y, and z positions of the manipulator's end-effector.

The virtual fixtures are preprogrammed on the remote computer, and generate the necessary forces to be displayed on the haptic device.

2.4.2 On-Board Computer

The laptop mounted on the Omnibot is a Linux system responsible for the rest of the control of the MMS. There are four peripherals connected to the laptop: the Gumstix for localization; the HCS12 for velocity control; the AVR microcontroller for the IR sensors; and the Powercubes. The first three systems are all connected via USB while the Powercubes are connected using the ESD USB-CAN bus adapter. ROS nodes are used to handle the connectivity between these devices. The nodes used are:

Cricket Reads the Cricket data from the Gumstix and posts it to a topic

OdometryDataFusion Combines the odometry and Cricket data to generate a pose estimate and publishes to a topic

CubeListener_automaticSwitching Communicates between ROS and "Teleop" program

joyListener Listens for motion commands to be posted to a topic and sends commands to the HCS12

Teleop Acts like a node but does not run in the ROS environment

Teleop controls the Powercubes and communicates with the ROS nodes and the remote workstation. Because the API for the Powercubes is not compatible with ROS, Teleop is not a ROS node. Communication between Teleop and the ROS nodes is achieved through IP socket communication. By communicating with the ROS node "CubeListener_automaticSwitching", it is able to receive the Omnibot and manipulator's pose. The Teleop program reads the joint positions of the Powercubes and calculates the xyz position of the end-effector using the forward displacement solution, Eq. (2.3) to (2.5), to send to the remote workstation. It also

receives velocity commands from the haptic device via the remote workstation and uses an inverse Jacobian matrix $\mathbf{J}_{arm}^{-1}$ to calculate the required joint velocities, which it then sends to the Powercubes using the USB-CAN interface and Powercube API.

2.5 Summary

The omnidirectional base "Omnibot" was combined with a Powercube manipulator to form the Omnibot MMS. The kinematic equations for the MMS were presented. To control the MMS, the haptic joystick by Novint called the "Falcon" was chosen for use with a number of computer systems to achieve teleoperation. An overview of the major systems of the Omnibot MMS was also shown.

Chapter 3

Command Strategy

There is a very important distinction to be made between control strategy and command strategy. The control strategy simply moves the robots given some desired trajectory, and facilitates communication. The command strategy is the way the system interprets the user commands, and performs appropriate actions. While various command strategies are presented herein, only one control strategy is used. The control architecture is explained in Section 3.1 and the remainder of Chapter 3 discusses command strategies.

3.1 Control Architecture

The control algorithm is executed among three different software nodes. These nodes perform three important functions:

User Interface Node Receives user input and displays virtual fixtures

Manipulator Control Node Controls the motion of the manipulator

ROS Communication Node Communicates between the Robot Operating System (ROS) framework and the Manipulator Control Node

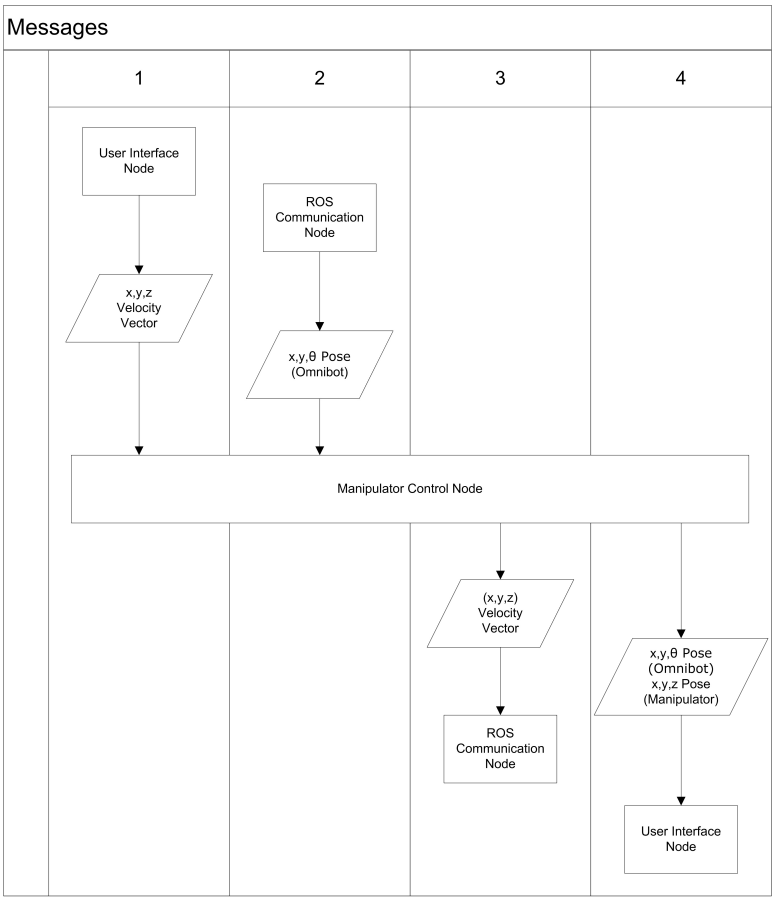


Figure 3.1: The Communication Pattern Between Nodes

Two forms of communication are used to facilitate the transfer of information between these nodes: Internet Protocol (IP) socket communication and the ROS framework. Packets of information are sent through each node in the loop, repeating until the program is halted.

Figure 3.1 shows the four message sequence sent between the three nodes when operating. Message 1 contains a velocity vector specified by the user. The velocity vector is proportional to the direction and magnitude the haptic device is displaced from the center of its workspace (the dead-band). Message 2 provides the manipulator control node with the Omnibot's pose. This information is important for converting from local to world coordinate frames, as well as when the command strategy switches modes. The manipulator control node then sends Message 3 containing the velocity vector to the ROS communication node.

The ROS communication node uses the velocity vector as a motion command for the Omnibot when in certain states. The fourth and final message is sent back to the user interface node where the message sequence begins again. Message 4 contains the pose of the Omnibot MMS, which is the x, y, and θ of the Omnibot and the x, y, and z of the manipulator. Using the pose data of the MMS, the user interface node calculates and displays the appropriate virtual fixtures. Each message is attached with a 4-digit code that represents the state of each node.

3.1.1 Initialization Nodes

Several initialization nodes are required for the Omnibot MMS to function properly. The majority of these nodes are responsible for localization. The first node involved with localization times the chirping of the two on-board Cricket devices. This node runs on the Gumstix computer embedded on the Omnibot. Initialization is done remotely using SSH protocols since the Gumstix computer does not have any user input devices. A ROS node called "Cricket" is then launched, and reads the serial data from the Gumstix computer to create an estimate of the position and orientation (pose) of the Omnibot. This estimate is posted to a ROS topic and is updated as new estimates are made. The node "OdometryDataFusion" reads the encoder data from each of the four wheels. The node subscribes to the Cricket topic and fuses the odometry data with the Cricket estimates to make the most accurate estimate of the pose it can. The corrected pose data is posted to a pose topic within ROS by the OdometryDataFusion node. Once launched, the Cricket and OdometryDataFusion node will continually update the pose estimates and make them available in a ROS topic.

The node that controls the movement of the Omnibot, called "joyListener", listens on a "joyChatter" topic for motion commands. When a motion command is posted to the topic,

joyListener sends the appropriate motion commands to the HCS12 velocity controller to move the wheels of the Omnibot.

3.1.2 ROS Interface Node

This node is a necessary bridge between the manipulator control node and the ROS framework. The reason the manipulator control node cannot work within the ROS framework is that the version of ROS used does not support the Application Programming Interface (API) necessary to control the manipulator. The ROS node provides localization data (from OdometryDataFusion) to the manipulator control node. The ROS node receives the data obtained from the haptic device (a velocity vector) by the user interface node via the manipulator control node. If necessary, this velocity vector is scaled appropriately and sent to the joyListener node to move the Omnibot.

3.1.3 User Interface Node

The user interface (UI) node has two main functions. The first function is to input information from the user, and the second is to display the appropriate haptic information. The UI node creates all the basic forces for the haptic device (such as the centering force) using the Novint API. The virtual fixtures surrounding the manipulator's workspace are handled by the UI node as well, and are dependent on the manipulator's pose, which is transmitted in Message 4 from the manipulator control node. The obstacle-avoidance virtual fixtures are also handled by the UI node with the Omnibot pose information included in Message 4.

3.1.4 Manipulator Control Node

The remaining functionality is provided by the manipulator control node. Aside from sending the motion commands to the manipulator, this node converts from world to local frames of reference, calculates the joint rotation speeds using the inverse Jacobian, and prevents the manipulator from reaching singularities.

The majority of the command strategy is implemented in the manipulator control node as well. The node uses the command strategy to decide when to send motion commands to the arm, or to the Omnibot through the ROS interface node. Depending on the pose of the MMS, the manipulator control node chooses the appropriate switching mode.

3.2 Manual Switching Command Strategy

The initial method of teleoperating the Omnibot Mobile-Manipulator System (MMS) consisted of using two input devices, one for the Omnibot and the other for the manipulator. In order to simplify this, a manual switching method was implemented. The switching implies that a single input device (the Novint Falcon) was used to control either the manipulator or the Omnibot. Since switching was manual, the operator was required to specify which device they wished to control. Using the buttons located on the Falcon's end-effector, the operator could select which device to control. By default, the operator controls the manipulator. If the operator wants to control the Omnibot, they would press and hold the centre button on the Falcon (see Figure 3.2). As with all other command strategies presented, a fourth degree-of-freedom (DOF) is simulated using the left and right buttons on the Falcon's end-effector. These buttons control the rotation of the Omnibot when pressed.

Figure 3.2: The Falcon End-Effector

3.3 Initial Automatic Switching Command Strategies

Manual switching is intended for expert level control. Someone who is very familiar with the characteristics of the Omnibot MMS can be considered an expert. The expert is able to complete a task using manual switching in the same time it would take them to complete a task using one of the automatic switching command strategies. The goal of automatic switching is to increase beginners to expert-level performance faster, and to reduce the amount of concentration required when expert-level performance is achieved. In other words, the goal is to make using mobile manipulators more intuitive for operators to use.

The first command strategy implemented for automatic switching was very similar to the work in [12]. When the manipulator reached the edge of its workspace, the manipulator would retract while the base moved simultaneously. In this command strategy, the end-effector would remain stationary relative to the operator as the arm is retracting. The base would also orient itself so that it would be facing the direction the arm was moving just before it started retracting. Once finished retracting, the operator would continue to control the arm. In this way, the operator never directly drives the base. The problem with this

command strategy is it takes a very long time to move any significant distance.

The next command strategy implemented was similar to the previous one, but once the arm retracted the operator would continue to control the base until they switched back to manipulator control. This was done through the use of a mode selector button, by tapping the button the system would switch states and return to manipulator control. The command strategy was compared to controlling the system using two joysticks. Figure 3.3 shows the experimental setup used for this testing. The operator was required to knock down two items at Target 1 and another two at Target 2. The time to perform each task was recorded, as well as the total distance traveled by the manipulator and the base. Table 3.1 shows the averaged results from the first round of testing. The single joystick method was slightly less efficient with respect to power consumption, but was much simpler to use (based on anecdotal evidence) and faster than using two joysticks. This initial experiment was enough proof to conclude that using a single joystick would be faster and simpler than using two joysticks. The results of these tests can be found in [65]. With this evidence, the next logical step was to improve the command strategy to increase operator productivity.

Table 3.1: Two Joystick Command Strategy vs. Single Joystick Command Strategy

	2 - Joysticks	1 - Joystick
Time (s)	54.4	49.0
Distance Traveled (Manipulator) (m)	1.60	1.95
Distance Traveled (Omnibot) (m)	9.72	9.50

Once it was established that it would be better to control the Omnibot MMS with a single joystick the task became finding the best command strategy to do so. The single joystick command strategy was certainly better than using two joysticks, but there are a variety of command strategies that can be used. The experiment in [66] was designed to test and compare two command strategies, one was named the “Virtual Wall” command strategy and the other the “Optimal Manipulability Pose” command strategy.

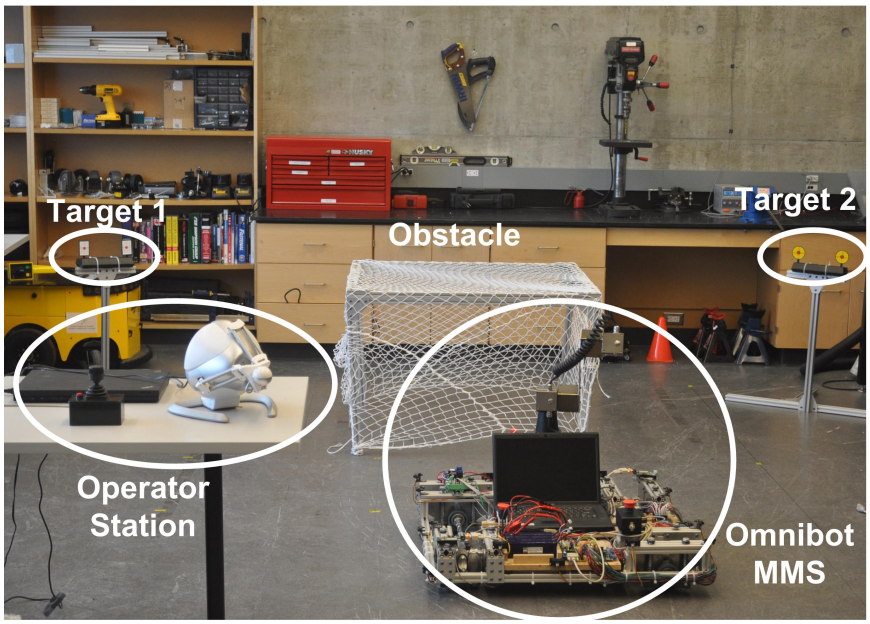


Figure 3.3: Experimental Setup With Targets, Obstacle, Operator Station, and Omnibot MMS Shown

3.3.1 Optimal Manipulability Pose

The Optimal Manipulability Pose (OMP) command strategy is very similar to the single joystick method presented in this section. In its initial state, the OMP command strategy allows the operator to control the manipulator exclusively. When controlling the manipulator, the operator uses velocity control. Velocity control is where the operator controls the velocity of the end-effector based on the position of the haptic device. In the centre of the haptic's workspace, there is a "dead-band" and when the haptic joystick is in the dead-band the manipulator will not move. To move the manipulator, the operator moves the joystick out of the dead-band. The direction from the centre of the dead-band to the joystick's end-effector is the direction the manipulator's end-effector will move. The further from the dead-band the joystick is, the faster the manipulator moves in the specified direction. There is a virtual fixture called the centering force always present on the haptic joystick. The centering force increases with magnitude as the joystick's end-effector moves further

away from the dead-band. The direction of the centering force is opposite to the specified direction of motion, always guiding the operator to the dead-band so they can easily stop the manipulator's motion.

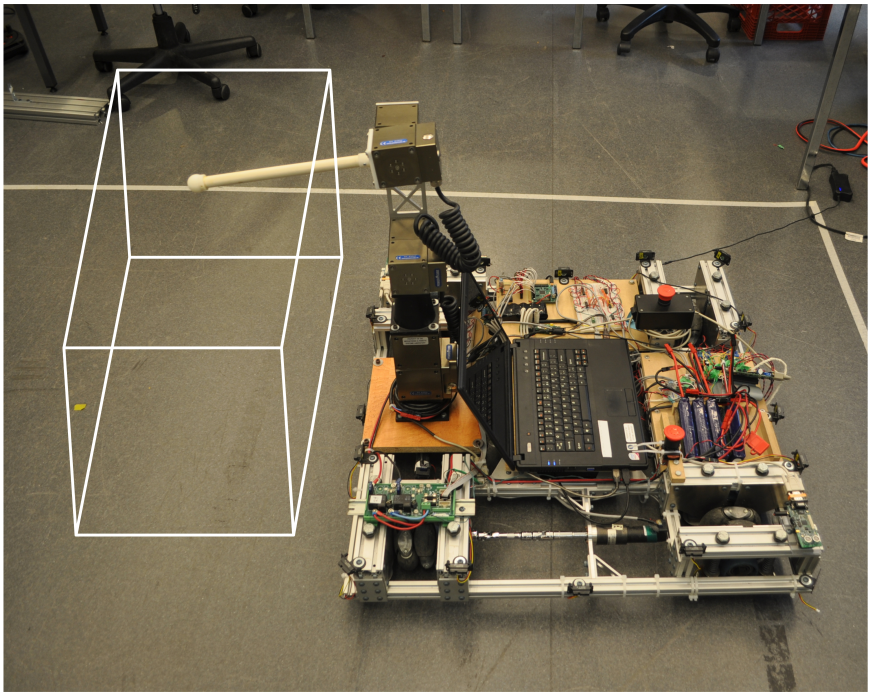


Figure 3.4: Virtual Box Surrounding Manipulator Workspace

A virtual box is placed around the manipulators workspace as shown in Figure 3.4. The virtual box exists as a virtual fixture as well. When the end-effector nears the virtual box, a repulsive force is felt on the haptic joystick. The repulsive force increases in magnitude the closer the manipulator gets to the edge of the virtual box. When the end-effector passes outside the box, the manipulator switches from manipulator control mode to base control mode. While in base control, the manipulator is retracted to an optimal manipulability pose. The OMP is user defined; in this case the manipulator is retracted so that the majority of the usable workspace is in front of the manipulator. This pose was chosen so that the operator has the most amount of usable space when switching back from base control to manipulator control, while still having some space to retract the arm if necessary.

When controlling the base, the operator uses velocity control to command the direction and speed. The rotation is controlled using the left and right buttons on the joystick's end-effector (see Figure 3.2). To switch back from base control to manipulator control, the operator presses the centre (mode selector) button for a short period. Along with the mode selector button, the operator can use a five second timeout to return to manipulator control by holding the joystick within the dead-band for that amount of time. This feature is rarely used because the majority of the testing evaluates time to complete. The base can also be controlled immediately without moving the manipulator out of the virtual box by pressing and holding the mode selector button. This feature is meant for use by experts, and during testing is considered a position correction.

3.3.1.1 Automatic Orientation

An automatic orientation feature is also available for the OMP command strategy. This function attempts to orient the Omnibot so it is facing the direction the operator requires. To do so, the command strategy assumes that when the manipulator breaks the virtual wall it is traveling in the direction of the next target. While in base control, the Omnibot will rotate until it reaches the desired orientation. Since the Omnibot is holonomic, and controlled in world coordinates, the rotation of the base does not affect the user when driving the Omnibot. Figure 3.5 shows how the Omnibot would orient itself when switching to base control mode.

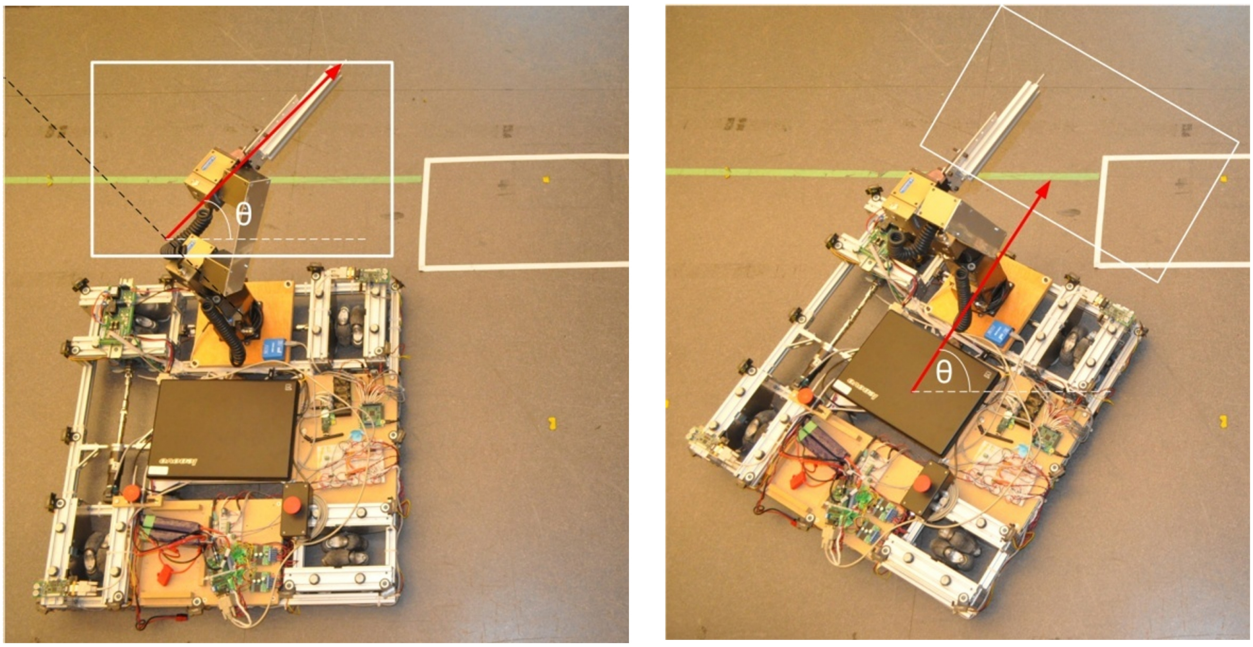


Figure 3.5: Automatic Orientation, Before and After Switching Control Modes

3.3.2 Virtual Wall

The virtual wall command strategy is identical to the OMP command strategy when controlling the manipulator. The manipulator is controlled using velocity control, there is a centering force, dead-band, and a virtual box with virtual fixture. The difference however occurs when the manipulator reaches the limit's of the virtual box. At the edge of the virtual box, the operator has two options: move the manipulator's end-effector outside the box, or back within the box. Moving the arm back within the box keeps the system in manipulator control. If the operator decides to move the end-effector outside of the box, the manipulator will not move. Instead, the base moves using the same velocity control scheme as in the OMP command strategy. It is important to note that the arm does not retract when using this command strategy. The command strategy always gives priority to the manipulator, so if at any point the operator moves the haptic joystick in a direction that would bring the manipulator away from the walls of the virtual box, then the arm will move. The base only moves when the operator is giving a command to move the manipulator outside the virtual box. The rotation buttons on the joystick are used to change orientation, and the mode se-

lector button can be used in the same way as the OMP command strategy. Use of the mode selector button is also considered as a correction.

3.3.3 Testing the Virtual Wall and Optimal Manipulability Pose Command Strategies

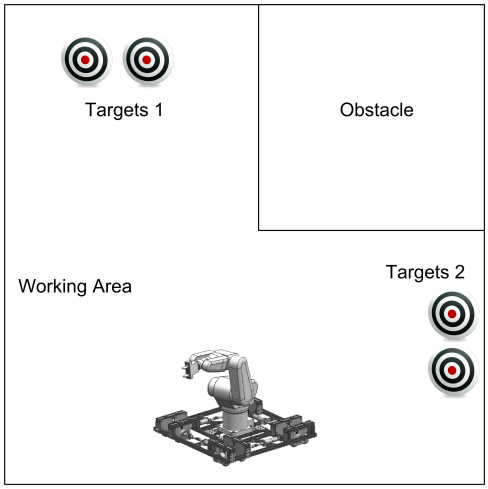


Figure 3.6: Command Strategy Testing Layout

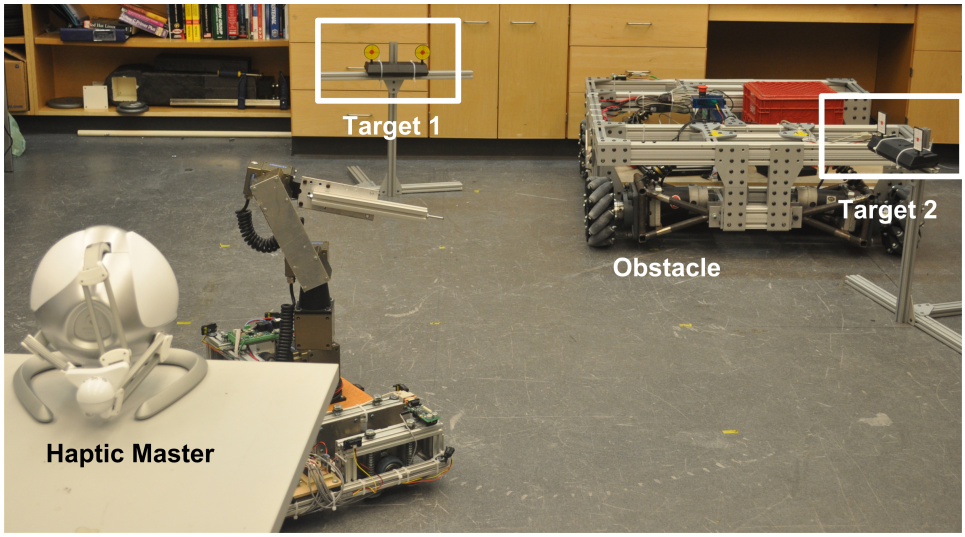


Figure 3.7: Testing Environment for Command Strategies

In order to compare the two command strategies, a simple task simulating pick-and-place

was created. The layout of the testing area is shown in Figure 3.6 and the testing environment can be seen in Figure 3.7. As with the initial test setup shown in Figure 3.3, the operator is required to knock down both items at Target 1, followed by both items at Target 2. The two command strategies were compared based on a number of criteria:

Completion Time The completion time measures the entire amount of time it took to complete the task. This is possibly the most important criteria because a goal of this work is to improve efficiency.

Omnibot Control This is the amount of time the user spent controlling the Omnibot. The Omnibot uses more power than the manipulator, so the less time the operator is controlling the Omnibot the more energy-efficient the command strategy is.

Manipulator Control The amount of time controlling the manipulator reflects how well the Omnibot was positioned. The better the base is positioned the less time is required for manipulator control.

Correction Time The correction time is the amount of time the operator had to control the Omnibot using the mode selector button. Any corrections necessary indicate the command strategy is not optimized.

Manipulator Movement The total distance traveled by the manipulator while in manipulator control.

Omnibot Movement The total distance traveled by the Omnibot.

Correction Distance The total distance traveled by the Omnibot using the mode selector button.

The last three criteria are to give context to the previous three. The amount of time spent controlling a specific subsystem must be coupled with the total distance traveled by that subsystem. With both pieces of information, one can now speculate whether the operator

was performing a slow and precise movement, or a quick movement over a large distance. The speed at which the operator moves these subsystems indicates the amount of concentration required. The averaged results of the testing is shown in Table 3.2 the complete testing results can be found in Appendix A.1.

Table 3.2: Averaged Results from Testing

	Virtual Wall	Optimal Manipulability Pose	
		Automatic Orientation	No Orientation
Completion Time (s)	74.73	92.24	61.80
Omnibot Control (s)	32.24	49.52	30.03
Manipulator Control (s)	42.48	42.72	31.77
Correction Time (s)	7.41	12.62	6.41
Manipulator Movement (m)	3.06	3.54	2.95
Omnibot Movement (m)	13.75	19.07	11.81
Correction Distance (m)	1.49	4.03	1.25

Conclusions The first conclusion made was that automatic orientation was not very helpful. Not only did the empirical results reflect that, but user testimony indicated so before testing was even completed. This feature was removed from the optimal manipulability pose command strategy and it drastically increased performance.

With no automatic orientation, the results from the two command strategies were much closer to each other, but the OMP command strategy was faster and more efficient. One reason the OMP command strategy was faster is when the operator wanted to switch modes, they could move the manipulator full speed towards the virtual wall. Using the virtual wall command strategy, the operator had to approach the virtual wall slowly as to avoid the base moving at full speed initially. Evidence to this fact is seen in the total time in base control, which only differs by 2 s, as opposed to the time in manipulator control which differs by 11 s.

Similar correction time and distance was seen on both command strategies. As well, similar manipulator movement distance was seen, indicating that both command strategies were able to position the base with equal amounts of accuracy. The virtual wall command strategy required more movement of the base, which leads to the conclusion that it is more difficult to drive the base using the virtual wall command strategy.

One vital piece of information that cannot be measured empirically is operator experience. When asked which command strategy was preferred by the operators mixed opinions were given. Most operators preferred the ability to make small repositions of the base using the virtual wall command strategy, but found it strenuous to drive long distances using it. It was concluded that overall the OMP command strategy was best, but the virtual wall command strategy was particularly useful when near the targets.

3.4 Virtual Fixtures

Several experiments were done using virtual fixtures on both the manipulator and the base in order to determine which would be best in the final command strategy. Since vision systems were not used, virtual fixtures pertaining to the environment had to be hard coded. In future work, these virtual fixtures could be created on the fly but this research merely tests their effectiveness.

3.4.1 Manipulator Control

Virtual fixtures, as well as command strategies specific to the manipulator, were studied in depth in [67]. In this work, a combination of velocity control and position control was used. Position control, also known as bilateral teleoperation, replicates the master's motions with

the slave's end-effector. Which ever way the operator moves the master, the slave performs the same motions. Position control provides a high level of accuracy, but its workspace is limited to the size of the master's workspace. To increase this workspace, a scaling factor can be used, but increased scaling leads to decreased accuracy. Velocity control, as previously explained, moves the slave's end-effector with velocity proportional to the masters displacement from its centre.

To optimize both workspace and accuracy, both control methods were used in a hybrid position-velocity command strategy. The command strategy would select which mode the user would control the manipulator is in based on the location of the manipulator's end-effector relative to predefined targets. When near to a target, the operator controls the manipulator using position control. When far enough away from the target, the user controls the manipulator using velocity control. Since the master will be displaced from its centre when the command strategy switches from position control to velocity control, the operator is required to "home" the master when switching modes. Homing the master is simplified using a virtual fixture similar to the centering force but with greater magnitude. Using position control a virtual spring pulls the master to match the location of the slave. Since the operator can move the master manipulator faster than the slave moves, the master and slave will not always be in the same position. Also, if the manipulator is colliding with an object and unable to move, the master and slave can be in two different positions. The spring force pulls the master towards the slave, so the operator can tell the difference in position between the master and slave. When in velocity control, only the centering force is present.

A pick-and-place task was designed to test the effect of virtual fixtures with this command strategy. Both attractive virtual fixtures and repulsive virtual fixtures were used. Figure 3.8 shows the experimental setup used to test the virtual fixtures and hybrid command strategy.

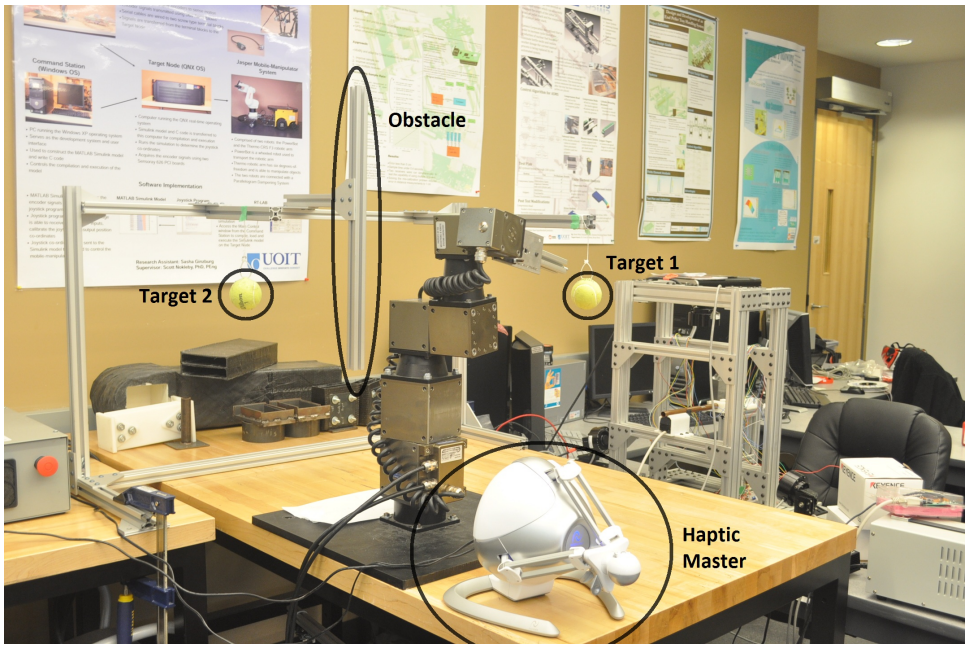


Figure 3.8: Testing Environment for Manipualtor Command Strategies

The users were required to simulate picking up an object by touching Target 1, then had to avoid the obstacle and reach Target 2 where the simulated placing of the object occurs.

Two virtual fixtures were used to improve the task efficiency. A repulsive virtual fixture helps the operator avoid the obstacle. When the end-effector approaches the obstacle, a repulsive force is felt on the haptic device pushing the manipulator away from the obstacle. An attractive force draws the operator's hand towards the target when sufficiently close and only while in position control.

Using four test subjects, each subject was required to perform the task eight times. Four runs were done with the virtual fixtures, and four without. The runs alternated between using the virtual fixtures and not using them, to minimize the impact of the learning curve on the results. Table 3.3 shows a summary of the results of the test.

The conclusion to be made about these results is that the virtual fixtures improved the per-

Table 3.3: Completion Time and Improvement

	Completion Time (s)		Improvement (%)
	With Virtual Fixtures	No Virtual Fixtures	
User 1	39.2	80.6	51.4
User 2	34.4	39.6	13.1
User 3	38.4	99.4	31.77
User 4	27.1	33.4	18.8

formance of the operators. It is worth noting that the operators who were able to perform the task with virtual fixtures quickly, did not suffer much when the virtual fixtures were removed. The operators that had trouble using the manipulator, experienced much more difficulty without the virtual fixtures. Accuracy was also improved with the use of virtual fixtures. Figure 3.9 shows the end-effector path for one run with virtual fixtures. The run took minimal time because the shortest path was taken and the obstacle was avoided. Figure 3.10 shows one run by an operator that struggled controlling the manipulator without virtual fixtures. In this run the operator took a long route to avoid the obstacle, and at one point collided with it.

3.4.2 Base Control

From the results in Section 3.4.1, one would assume that attractive and virtual fixtures would be helpful when controlling the base as well. To test this assumption, virtual walls were set up in the testing area to simulate real walls, and the Omnibot MMS was driven through it. Since the position of the walls were hard coded, the accuracy of the distance between the walls and the Omnibot is dependent on the accuracy of the localization system.

Because of the way the localization system fuses odometry and Cricket data to estimate the Omnibot's pose, a significant loss of accuracy (up to 15 cm) is experienced. Because of this, when repulsive virtual wall fixtures were used, the virtual fixtures were felt even

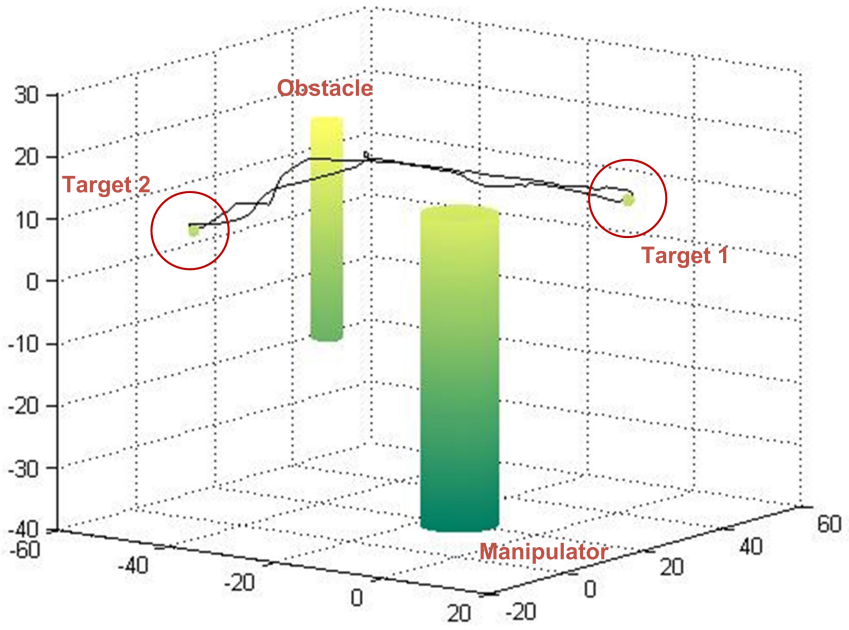


Figure 3.9: Sample Run Using Virtual Fixtures

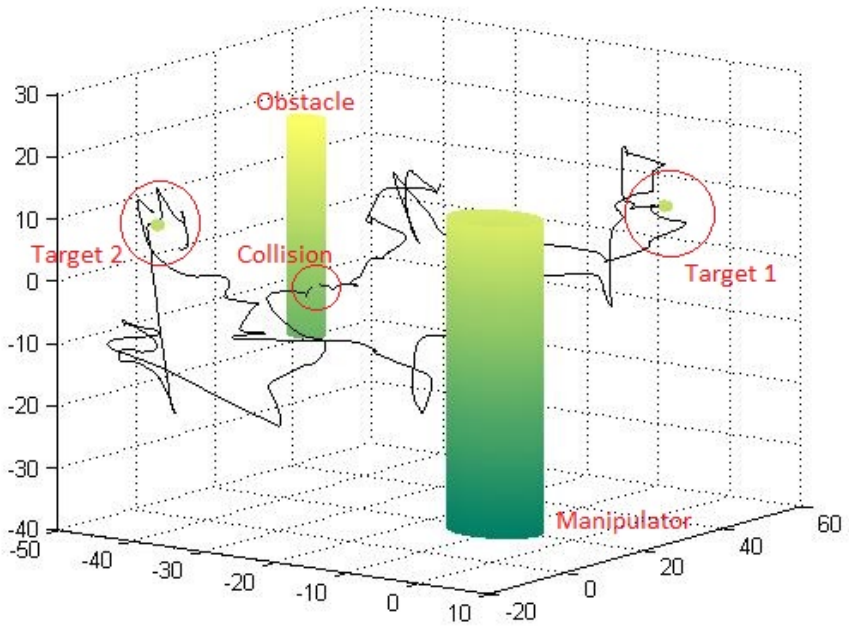


Figure 3.10: Sample Run Without Virtual Fixtures

when sufficiently far enough away from the wall that they should not have been felt. For this reason a different type of virtual fixture was used. The virtual fixture did not create any haptic forces, but acted as a "forbidden region" virtual fixture. Using this type of virtual fixture, the Omnibot will only drive in directions away from a virtual wall when touching one. This way, the Omnibot does not hit any virtual walls and the operator does not feel any erroneous forces.

3.5 Final Command Strategy

Using the results from the preliminary testing, the final design for the Omnibot MMS command strategy was created. The design uses a combination of virtual fixtures, velocity control, and some of the command strategies tested. In order to take advantage of both the Optimal Manipulability Pose (OMP) command strategy as well as the Virtual Wall command strategy, either can be used depending on the proximity of the Omnibot MMS to its targets. When close to the targets, the MMS uses near-target manipulation mode. When a certain distance away from the targets, the MMS uses off-target manipulation mode. When not controlling the manipulator and driving the base, the MMS uses transportation mode. Through the use of automatic switching, the MMS switches between these modes seamlessly and autonomously.

3.5.1 Transportation Mode

Transportation mode is used when the operator needs to drive the MMS to a location the manipulator cannot reach. The Omnibot is controlled by the haptic master in the style of velocity control. There is a dead-band in the centre of the master's workspace, and a centering force that pulls the master's end-effector towards the dead-band. To move the Omnibot,

the operator moves the joystick out of the dead-band in the direction they wish the Omnibot to drive. The further from the dead-band the joystick is, the faster the Omnibot moves. To rotate the Omnibot, the left and right buttons on the joystick are used.

3.5.2 Off-Target Manipulation Mode

When the user begins operating the Omnibot, the system is in off-target manipulation mode. In this mode, the operator controls the manipulator's end-effector in velocity control. In the same way velocity control of the Omnibot works, so does velocity control of the manipulator work. The dead-band is used, and the centering force draws the joystick towards it. The velocity of the manipulator is proportional to the displacement from the dead-band. When controlling the manipulator, the operator commands it in world coordinates. For example, when the operator moves the joystick away from themself the manipulator also moves away from themself regardless of the orientation of the Omnibot.

When controlling the manipulator in off-target manipulation mode, virtual fixtures are felt at the limits of the manipulator's workspace, pushing the manipulator away from its joint limits. If the operator still moves the manipulator towards the workspace limits, the arm will retract to a predefined pose and enter transportation mode. Off-target manipulation mode closely resembles the OMP command strategy tested.

The system remains in off-target manipulation mode while the Omnibot is further than 15 cm from the target. When closer than 15 cm, the MMS operates in near-target manipulation mode.

3.5.3 Near-Target Manipulation Mode

When less than 15 cm from the target, the MMS enters near-target manipulation mode. This manipulation mode works in the same way the virtual wall command strategy functioned. The manipulator is controlled just as it was in off-target manipulation mode. When the manipulator is moved to its workspace limit, the base moves in that direction. The arm does not retract, and the system does not enter transportation mode. After moving the base if the operator moves the arm away from its workspace limits, the arm will move and not the base. If the operator uses near-target manipulation mode to move the base further than 15 cm from the target, the MMS will switch to off-target manipulation mode and then retract the arm as it enters transportation mode. The reason the MMS switches right into transportation mode is that at the moment the MMS switches to off-target manipulation mode, the operator is already commanding the manipulator to move outside its workspace, which causes off-target manipulation mode to switch to transportation mode.

3.5.4 Automatic Switching

The system automatically switches between the three modes autonomously. It switches between near-target and off-target manipulation modes based on its location. In future work, the locations of the targets can be set dynamically, but that is outside the scope of this work and the target locations were preprogrammed. The automatic switching between off-target manipulation mode and transportation mode is done based on the manipulator's position and workspace limits.

3.5.5 Virtual Fixtures

When in either of the manipulation modes, the virtual fixtures present represent the limits of the manipulator's workspace. Virtual walls exist around the workspace as repulsive virtual fixtures to alert the operator when they are approaching the limits. The force of the virtual fixture is proportional to the distance from the limit of the workspace and is defined as:

$$F_x = -\left(0.02 - \left(X_{wall} - X_{arm}\right)\right) * 1000 \tag{3.1}$$

$$F_y = -\left(0.02 - \left(Y_{wall} - Y_{arm}\right)\right) * 1000 \tag{3.2}$$

where F_x and F_y are the forces generated on the haptic device and expressed in Newtons, the value 0.02 is the distance in meters from the virtual wall at which the operator begins to feel forces, X_{wall}/Y_{wall} and X_{arm}/Y_{arm} are the X and Y coordinates of the virtual wall and the manipulator's end-effector, respectively, in meters, and the value $1,000$ is a unitless scaling factor.

An additional virtual fixture is present but not felt. It is a forbidden region virtual fixture and prevents the Omnibot from hitting walls. If the Omnibot is touching a wall, the MMS will not drive towards the wall while in transportation mode. If the operator commands the Omnibot to move towards a wall when touching the wall, motion commands are ignored. To prevent accidental damage, these walls are simulated by markings on the floor of the working environment. Just like the locations of the targets that will eventually be created dynamically, the locations of the walls are preprogrammed for this work.

3.6 Summary

This chapter presented both the control architecture and the command strategy used to operate the Omnibot MMS. The nodes responsible for the control architecture were outlined and explained.

The implementation of the control strategy on the Omnibot MMS test-bed was done mainly using three software nodes. Initialization nodes control the localization and mobilization of the Omnibot MMS. The ROS interface node allows communication between the manipulator control node and the initialization nodes. The manipulator control node controls the arm of the MMS, and serves as the "brain" of the MMS in which the command strategy is implemented on. The user interface node receives the desired motions from the user via the haptic input device, and displays the virtual fixtures though the same haptic device.

A number of command strategies were presented along with testing results. Initially, the Omnibot MMS was controlled using two joysticks. Manual switching allowed the operator to control both the arm and the base using a single joystick. Using the Omnibot MMS to autonomously switch modes allowed the testing of various command strategies employing automatic switching. The two most successful automatic switching command strategies were the optimal manipulability pose and virtual wall command strategies, the former slightly more than the latter.

Before the final command strategy was implemented, virtual fixtures were tested on the arm and the base individually. Since the arm had greater accuracy in its pose estimates, the haptic virtual fixtures were quite helpful. Due to the poor accuracy of the Omnibot localization, the haptic virtual fixtures were not as successful so non-haptic virtual fixtures were used.

The final command strategy used in this work was presented, explaining the three modes of operation: near-target manipulation mode, off-target manipulation mode, and transportation mode. The automatic switching and virtual fixtures were discussed as well. In Chapter 4 the final command strategy will be tested and evaluated.

Chapter 4

Testing and Results

The final testing of the single joystick command strategy proves two important things. By implementing the command strategy, it is proven to be possible to control the Omnibot Mobile-Manipulator System (MMS) using a single joystick. Not only is the final test a proof-of-concept, but it proves that the new command strategy can successfully perform the required task. In order to test the effectiveness of the command strategy versus the previous dual joystick command strategy, a testing setup was used with a number of volunteers to compare the new command strategy against the old one.

4.1 Testing

4.1.1 Experiment Layout

Figure 4.1 shows the layout of the test set-up used. In Figure 4.2 the actual testing area is shown. The white lines represent walls in the workspace of the Omnibot MMS. Since the location of the walls are preprogrammed, and to prevent damage to the robot, the actual walls are not necessary as the visual representation of them is sufficient. The targets simulate two pick-and-place tasks. Knocking over the first item simulates picking up an item

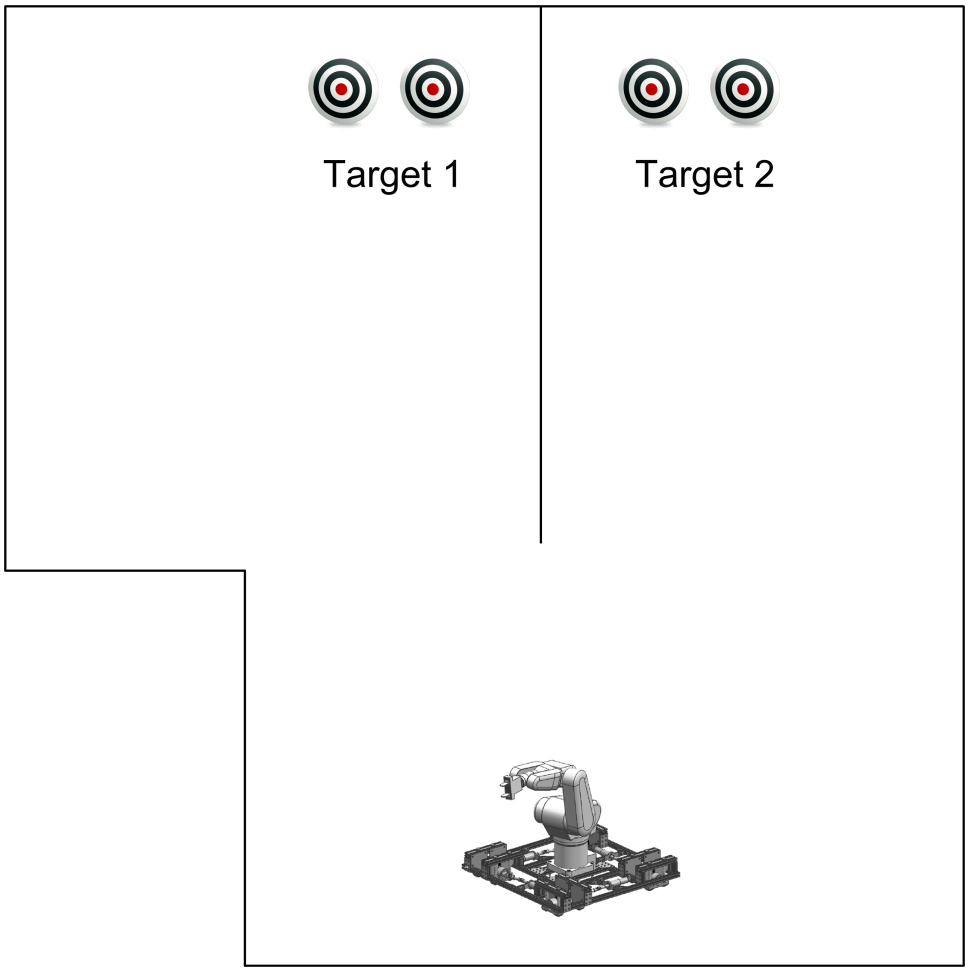


Figure 4.1: Testing Area Layout

and knocking over the second one simulates placing it (see Figure 4.3). This pick-and-place action is performed a second time at a second location, called Target 2.

4.1.2 Testing Strategy

The task required of the operator tests a number of operator skills. First, the operator must drive the Omnibot to Target 1. This briefly tests the navigation skills of the operator. Once at Target 1, the operator must position the Omnibot in a suitable location for the arm to reach the target. This action tests the operator's fine control of the Omnibot base when positioning it. Once positioned, the operator's skill using the manipulator is tested as they knock down

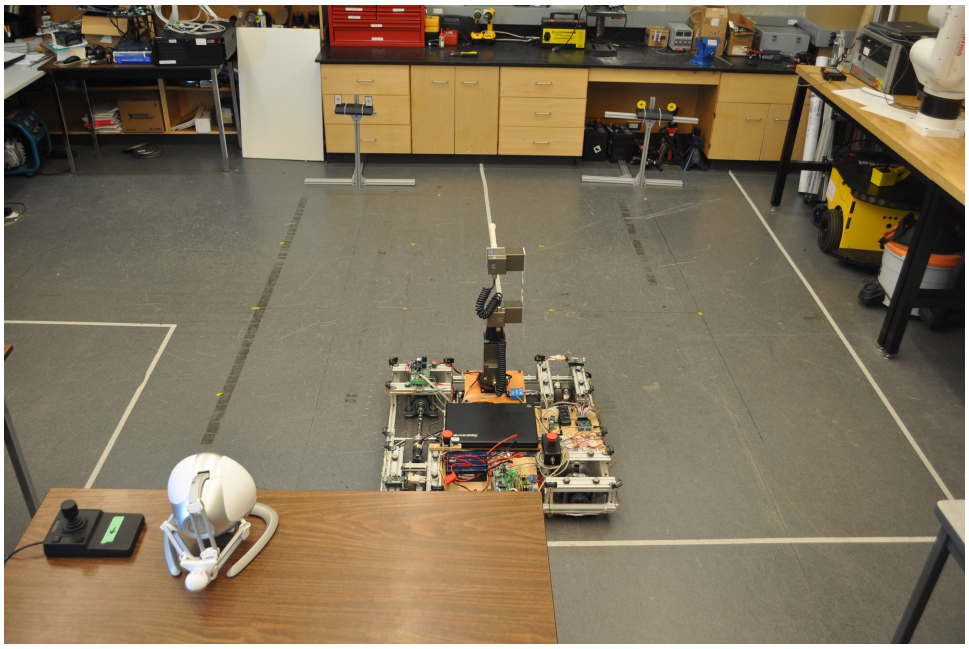


Figure 4.2: Testing Area

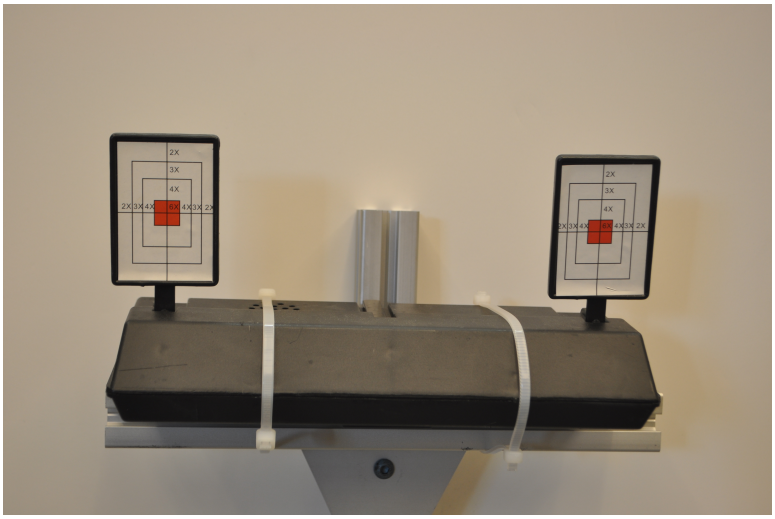


Figure 4.3: Two Pick-and-Place Items at Target 1

the two items at Target 1. When driving from Target 1 to Target 2, the operator's navigation skills are further tested as they avoid the walls on their way to Target 2. When arriving at Target 2, the Omnibot positioning and manipulator control are tested again. Navigation is tested one more time as the operator returns the Omnibot MMS to the original start position.

As discussed in Section 4.2, there is a very evident learning curve when using the new com-

mand strategy. In order for this learning curve not to affect the comparison between the two command strategies, each test run alternated between the single joystick and dual joystick command strategies.

The final testing was performed with five different operators, four novices and an expert. Each test subject was required to do ten test runs, alternating between the two command strategies. At a frequency of 50 Hz the pose of the Omnibot MMS and a timestamp is recorded. The pose contains the x and y position of the Omnibot, the orientation of the Omnibot, and the x, y, and z position of the manipulator's end-effector.

4.2 Results

After recording the data from each of the operator's test runs, additional analysis was performed. The average completion time for each run was computed, along with the improvement in percentage of their final run. The total distance traveled by the Omnibot, the time spent driving the Omnibot, and the average speed of the Omnibot were calculated. Also, the total distance traveled by the arm, time spent manipulating the arm, and average speed of the arm were calculated. These results are shown in Table 4.1.

Table 4.2 is the averaged results from Table 4.1, as well as the percentage of improvement of the final run.

Finally, the learning curve was calculated for the four novice users. These curves are shown in Figures 4.4 to 4.7.

Table 4.1: Experimental Results

		Run	Completion Time (s)	Omnibot Travel (m)	Omnibot Time (s)	Arm Travel (m)	Arm Time (s)	Omnibot Speed (m/s)	Arm Speed (m/s)
User 1	Single Joystick	1	133.74	15.64026359	54.37	4.752432	79.37	0.287663	0.059877
		2	129.011	16.0081	39.444	3.934803	89.567	0.405846	0.043931
		3	91.533	17.0336	42.946	3.010357	48.587	0.39663	0.061958
		4	96.232	15.3371	39.808	3.815849	56.424	0.385279	0.067628
		5	82.915	15.5695	42.139	3.039733	40.776	0.36948	0.074547
	Two Joysticks	1	80.612	19.1558	27.33	1.387832	53.282	0.70091	0.026047
		2	99.413	19.7375	36.967	1.903366	62.446	0.533925	0.03048
		3	84.088	17.4719	33.545	1.49499	50.543	0.520851	0.029579
		4	110.372	18.4479	54.502	2.547398	55.87	0.338482	0.045595
		5	82.045	16.6503	29.329	1.391311	52.716	0.56771	0.026393
User 2	Single Joystick	1	116.99	24.1051	67.239	3.22668	49.751	0.358499	0.064857
		2	125.155	20.4341	78.932	4.327533	46.223	0.258883	0.093623
		3	80.643	19.6276	45.931	3.80933	34.712	0.427329	0.109741
		4	55.121	17.3388	34.514	2.477165	20.607	0.50237	0.12021
		5	46.032	15.3248	30.869	2.283855	15.163	0.496449	0.15062
	Two Joysticks	1	230.493	24.3332	38.452	1.315723	192.041	0.632821	0.006851
		2	119.394	16.9296	21.458	1.033711	97.936	0.788965	0.010555
		3	59.758	16.1568	22.524	0.921761	37.234	0.717315	0.024756
		4	72.62	17.0865	27.112	1.3889	45.508	0.630223	0.03052
		5	55.903	19.6092	22.182	0.952041	33.721	0.884014	0.028233
User 3	Single Joystick	1	177.587	23.5925	89.527	6.701303	88.06	0.263525	0.076099
		2	63.828	13.0475	33.838	2.628552	29.99	0.385588	0.087648
		3	115.795	17.4314	59.982	4.279703	55.813	0.290612	0.076679
		4	80.744	13.9728	37.647	4.170967	43.097	0.371154	0.096781
		5	67.576	18.0269	45.467	2.894011	22.109	0.396485	0.130897
	Two Joysticks	1	99.569	17.5112	50.334	1.961929	49.235	0.3479	0.039848
		2	89.281	16.9665	39.722	2.150828	49.559	0.427132	0.043399
		3	95.248	19.8759	32.907	1.371626	62.341	0.604002	0.022002
		4	78.631	15.8143	20.774	1.29386	57.857	0.761257	0.022363
		5	70.608	16.5219	26.304	1.904357	44.304	0.628115	0.042984
User 4	Single Joystick	1	134.551	18.2452	60.634	5.545257	73.917	0.300908	0.07502
		2	79.955	17.7260	43.042	3.808524	36.913	0.411831	0.103176
		3	105.709	18.0271	49.346	4.463415	56.363	0.365322	0.079191
		4	66.01	13.5344	33.501	3.161972	32.509	0.404001	0.097265
		5	44.963	12.6184	27.649	1.71398	17.314	0.456381	0.098994
	Two Joysticks	1	136.8973	21.4055	44.245	1.442356	92.652	0.483789	0.015568
		2	114.9237	18.6283	41.021	1.747573	73.902	0.454113	0.023647
		3	83.75467	17.5844	49.590	1.260496	34.164	0.354592	0.036895
		4	98.16267	18.2487	42.777	1.71239	55.385	0.426595	0.030918
		5	75.274	18.7217	28.149	1.184888	47.125	0.665095	0.025144
User 5 (Expert)	Single Joystick	1	90.79	18.3591	50.423	2.393099	40.367	0.364103	0.059284
		2	59.791	16.9917	36.39	2.367067	23.401	0.466936	0.101152
		3	52.797	14.0445	30.337	1.954985	22.46	0.46295	0.087043
		4	45.776	13.8665	28.942	1.733846	16.834	0.479116	0.102997
		5	44.963	12.6184	27.649	1.71398	17.314	0.456381	0.098994
	Two Joysticks	1	99.587	20.7275	66.955	1.623513	32.632	0.309574	0.049752
		2	125.964	19.2177	64.639	2.305643	61.325	0.297309	0.037597
		3	107.418	19.1245	92.703	1.364736	14.715	0.2063	0.092745
		4	111.496	19.2117	46.719	1.200872	64.777	0.411219	0.018539
		5	87.874	19.9057	32.936	1.211312	54.938	0.604376	0.022049

Table 4.2: Averaged Results From Table 4.1

		Completion Time	Omnibot Travel	Omnibot Time	Omnibot Speed	Arm Travel	Arm Time	Arm Speed	Improvement
User 1	Single Joystick	106.6862	15.9177	43.7414	0.3639	3.7106	62.9448	0.05895	-1.0639
	Two Joystick	91.306	18.2927	36.3346	0.5034	1.7449	54.9714	0.03174	
User 2	Single Joystick	84.7882	19.3661	51.497	0.3760	3.2249	33.2912	0.09686	17.6573
	Two Joystick	107.6336	18.8230	26.3456	0.7144	1.1224	81.288	0.01380	
User 3	Single Joystick	101.106	17.2142	53.2922	0.3230	4.1349	47.8138	0.08647	4.2941
	Two Joystick	86.6674	17.3379	34.0082	0.5098	1.7365	52.6592	0.03297	
User 4	Single Joystick	86.2376	16.0302	42.8344	0.3742	3.7386	43.4032	0.08613	40.2675
	Two Joystick	101.8024	18.9177	41.1568	0.4596	1.4695	60.6456	0.02423	
User 5 (Expert)	Single Joystick	58.8234	15.1761	34.7482	0.4367	2.0325	24.0752	0.08442	48.8324
	Two Joystick	106.4678	19.6374	60.7904	0.3230	1.5412	45.6774	0.03374	

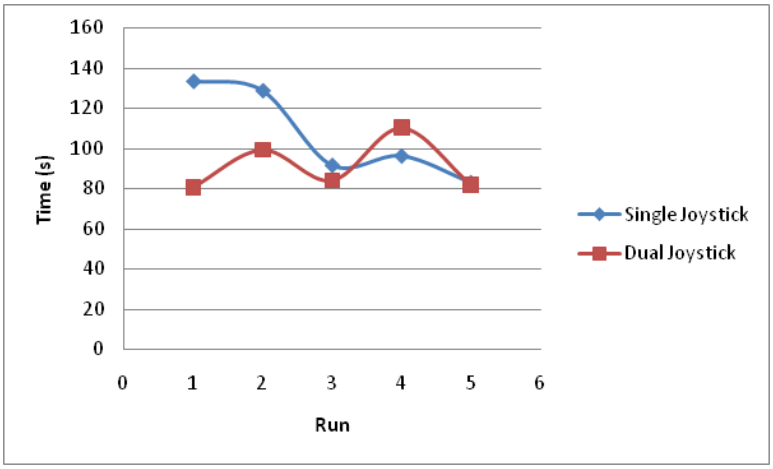


Figure 4.4: Learning Curve for User 1

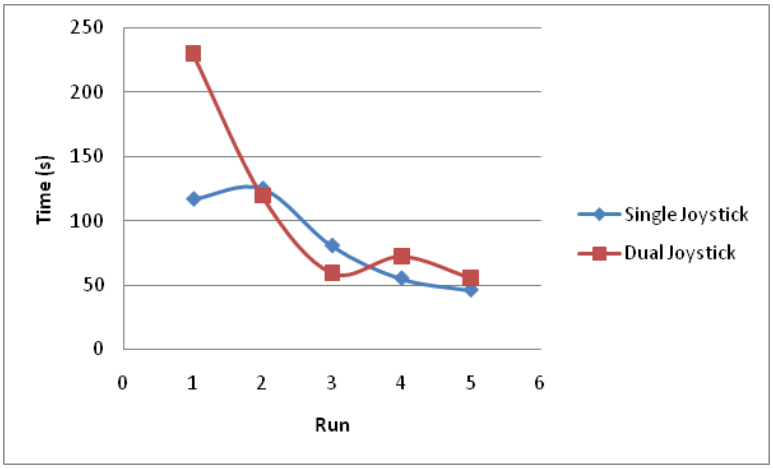


Figure 4.5: Learning Curve for User 2

4.3 Analysis of Results

The results can be analyzed based on two sets of data. First, all the data presented in Section 4.2 can be analyzed to judge the command strategy based on performance. Analyzing

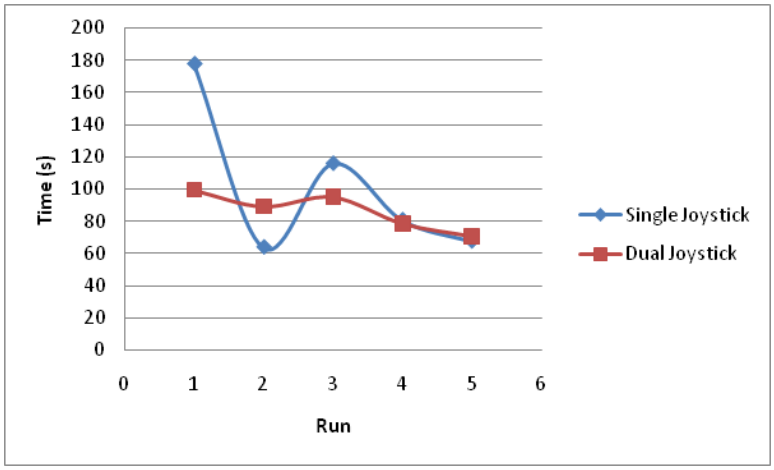


Figure 4.6: Learning Curve for User 3

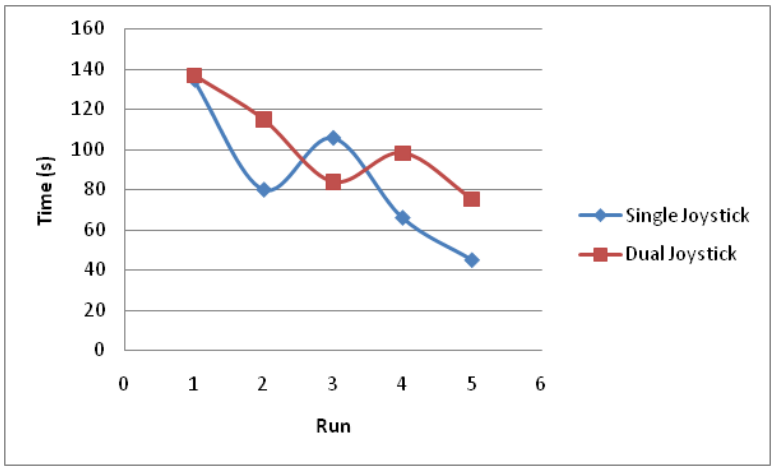


Figure 4.7: Learning Curve for User 4

the command strategy using strictly the empirical data only paints half the picture, user testimony must also be taken into account when comparing command strategies.

4.3.1 Empirical Data

4.3.1.1 Learning Curve

When inspecting the learning curves from the novice users (see Figures 4.4 to 4.7), a clear downward slope is seen. The completion time in the final run of the novice users ap-

proached the completion time of the expert in just five test runs. One can assume that in a few more test runs, the novices will perform at expert levels. The learning curve of the single joystick command strategy is very short, and in some cases novice operators were performing as fast or faster than the expert after five runs. This indicates that though the command strategy is difficult at first, operators can learn it quickly. When comparing the learning curve of the single joystick command strategy versus the dual joystick command strategy, one can note that the learning curve of the dual joystick command strategy is less pronounced than the single joystick command strategy. Based on the two learning curves, the assumption that the operators using the dual joystick command strategy will not continue to improve as much as they will using the single joystick command strategy can be made. This assumption is further validated when comparing the novice results to the expert results, where the learning curve has leveled off.

4.3.1.2 Completion Time and Improvement

The percentage of improvement noticed in the test subjects was quite varied among the novices, but very noticeable in the expert runs. If testing were to continue, the novices would reach expert status quickly and a more prominent increase in efficiency when using the single joystick command strategy would become evident. When inspecting performance at the end of the learning curve all operators performed the same or better than using two joysticks.

4.3.1.3 Omnibot Control

In most cases, the total distance traveled by the Omnibot was shorter using the new command strategy than the old one. This indicates that the single joystick method is more efficient when driving the Omnibot. The total amount of time driving the Omnibot was

much longer for the novice test subjects. This is partially due to the single joystick command strategy having a lower maximum speed while driving the Omnibot compared to the dual joystick method. Since the command strategy aims to improve not only performance but accuracy as well, it was decided to have a lower maximum speed and therefore a more precise range of speeds.

Another reason for the extended driving time is because of the localization system and the forbidden region virtual fixtures. The localization system sometimes has poor accuracy, especially when at the limits of the Omnibot MMS workspace. The accuracy is so poor at the workspace limits, that given the position in Figure 4.8, the localization system may assume the Omnibot MMS is in the position in Figure 4.9 or the position in Figure 4.10. This can be very confusing to the operator, because the robot will not drive towards a wall if the localization system assumes it's touching one. When a mistake like this occurs, the operator does not know whether the localization system mistakenly assumes the Omnibot MMS is in the position shown in Figure 4.9 or Figure 4.10. Not knowing which position the localization system is returning, the operator does not know which wall to drive away from. Because of this, the novice operators spent significantly more time driving the Omnibot base.

The localization error caused an increase in task completion time for another reason: when switching from off-target to near-target manipulation mode, the event may occur too early or too late. If the switch happens too early, the operator has to extend the arm to its workspace limits and reposition the base. If it happens too late, the operator will either collide with the target or have to manually switch back to manipulation mode using the mode selector button.

The above errors could be minimized by improving the localization system accuracy. Im-

provements to the localization system were beyond the scope of this thesis.

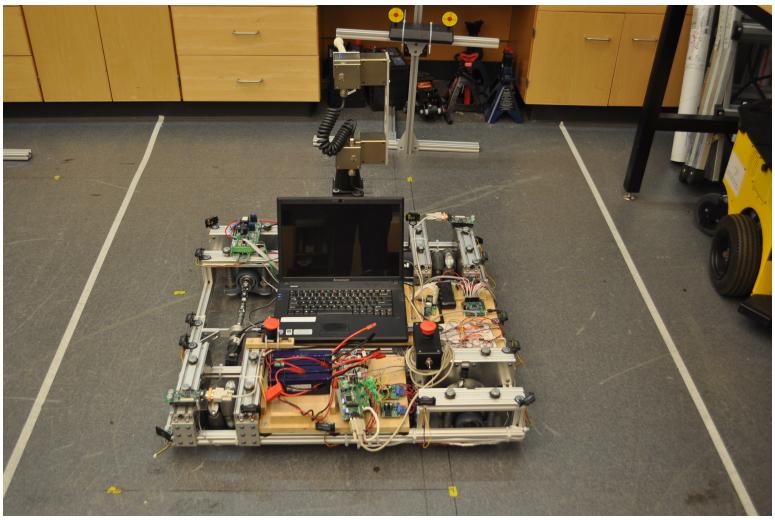


Figure 4.8: Omnibot MMS in its True Position

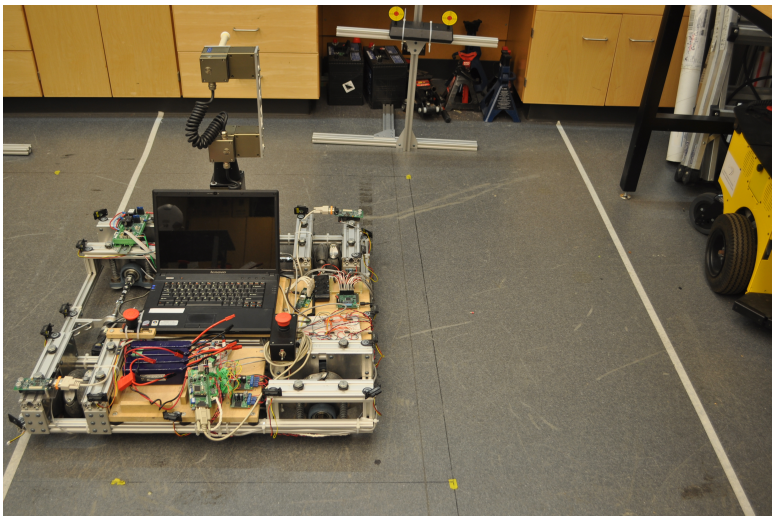


Figure 4.9: Incorrect Localization Estimate

4.3.1.4 Manipulator Control

The total distance traveled by the arm is much longer in the single joystick command strategy because in order to switch from manipulation mode to transportation mode, the operator is required to extend the manipulator to the limits of its workspace. Even though more dis-

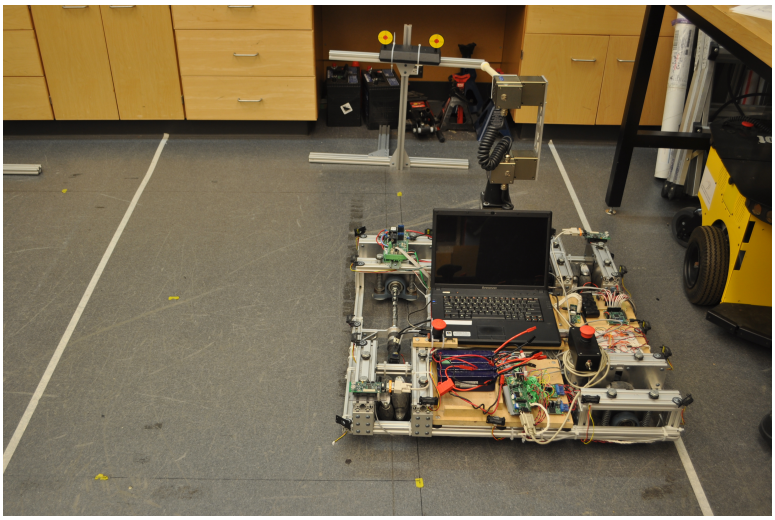


Figure 4.10: Incorrect Localization Estimate

tance was traveled by the manipulator, the majority of operators needed less time when using the manipulator indicating that the positioning of the Omnibot base near the targets was more accurate than using the dual joystick method.

4.3.2 Anecdotal Evidence

Each of the operators were asked their opinion of the new command strategy. The operators said that it was convenient to have the MMS automatically switch modes when near to the targets. The virtual wall fixtures around the manipulator's workspace were helpful as well, especially when trying to hit the targets at the limits of the manipulator's workspace. The near-target manipulation mode was useful when operators needed to reposition the base, but when the task was completed it was un-helpful to have to use near-target manipulation to move far enough away to enter off-target manipulation mode and subsequently transportation mode. The forbidden region virtual fixtures did prevent collisions with the walls, but often malfunctioned and made it difficult to drive the Omnibot due to the localization errors. Some operators felt that the speed limiting was not necessary when using the single joystick command strategy.

Using one joystick the operators had a free hand to perform other tasks, increasing their functionality. Using two joysticks simultaneously was too difficult for concentration and the operators only used one of the two joysticks at a time. The operators agreed that it was less stressful and required less concentration when using the single joystick method.

4.4 Sources of Error

Aside from the localization errors discussed, there were few sources of error in this experiment. The localization error can be extended to the Omnibot travel data, and therefore that data may not be entirely accurate. The position data of the manipulator is very accurate because it is calculated using the forward displacement solution based on the joint angles. The positional accuracy of the joints and the encoders are excellent due to the high quality of the Powercube modules. The time measurements were calculated based on the computer's system time, and is as accurate as the computer system itself.

4.5 Summary

An experimental setup was constructed to test the two command strategies. The first strategy, using two joysticks, is the traditional method of controlling MMS. The new strategy proposed, uses a single joystick to control all 6-DOF (degrees-of-freedom) of the MMS. To test the effectiveness of the new command strategy, a number of operators with no experience were asked to complete the tasks in the experiment setup. The results of the novices were compared to the results of an expert operator to asses the speed at which the operators learn the new command strategy.

After just five runs, the novice operators were able to perform as well or better than the two joystick method when using one joystick. Improvement was seen in both command strategies, but the learning curve indicated more improvement using the single joystick command strategy. The operators were more efficient with the single joystick command strategy, both in base movement and manipulator control time. The overall time to completion for the single joystick method was not as optimal as it could have been, due to inaccuracies in the localization system.

The operators were asked about their experience with the new command strategy using only one joystick. All of them agreed that they liked the single joystick better than using two. The operators noted that they felt very comfortable with the single joystick and they felt they could control the Omnibot base with more speed than the command strategy permitted. Most said that using two joysticks was difficult to concentrate on, and it was nearly impossible to control both simultaneously. When using one joystick, the operators had a free hand to perform another task.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

A novel command strategy for Mobile Manipulator Systems (MMS) has been developed. This command strategy simplifies the control of a MMS by requiring only one input device to control all of its degrees-of-freedom (DOF). The use of virtual fixtures assist the operator in performing tasks using the MMS. To test the command strategy, it was implemented on the 6-DOF holonomic MMS called the Omnibot MMS. Though only tested on one MMS, the command strategy can be easily applied to a variety of MMS with more than 6-DOF.

The command strategy has several key features. The most visible feature is that it uses only one joystick. The single joystick has 3-DOF, but controls all 6-DOF of the MMS without using mathematical redundancy resolution. Another feature not visible is the command strategy uses virtual fixtures to improve task performance. The command strategy is a teleoperation command strategy, meaning that the operator can control the MMS at any distance in real-time, provided that the distance does not exceed real-time communication speeds (such as the distance between Earth and Mars).

The single joystick command strategy operates in three unique modes, two dedicated to controlling the manipulator and one mode for driving the base. The mode for driving the base, called transportation mode allows the operator to drive the base in world coordinates. Driving the base in world coordinates means that the operator can focus on the direction they want the Omnibot to travel, regardless of its orientation. In the two manipulation modes, the operator also controls the manipulator in world coordinates to simplify teleoperation of the manipulator. A repulsive virtual fixture called the virtual wall borders the workspace of the manipulator. When the manipulator's end-effector nears this virtual wall, the operator feels a repulsive force. This repulsive force is used to assist the operator in maintaining the end-effector within the manipulator's workspace. When the operator tries to move the manipulator outside of its workspace, one of two actions occurs depending on the manipulation mode. In near-target manipulation mode, the base will move in the direction the manipulator was moving, and return to manipulator control when the operator ceases commanding the manipulator in a direction outside the manipulator's workspace. If the system is in off-target manipulation mode, the arm will retract to an optimal pose within its workspace and then switch to transportation mode. The system automatically selects which manipulation mode to use based on its proximity to the targets.

Forbidden region virtual fixtures were used to protect the MMS from damage. The forbidden region virtual fixtures prevent the MMS from driving into walls by ignoring commands that would do so. Repulsive virtual fixtures were implemented, but not used, on the walls to help the operators navigate away from them.

Novice operators were asked to perform two simulated pick-and-place tasks with the Omnibot MMS and their performance was compared to that of an expert. After five runs the novice operators' performance approached that of an expert operator, and had met or surpassed their performance using two joysticks. Due to speed limiting and localization issues,

the amount of time it took to drive the Omnibot was increased, but upon further investigation the total distance driven by the Omnibot was decreased indicating an improvement in efficiency and leading to the conclusion that with increased speed and localization accuracy the time driving the Omnibot would be decreased as well. The localization error does not affect the two joystick command strategy and therefore no improvement in performance would be noticed, further emphasizing the strength of the new command strategy. Based on user testimony, the new command strategy was a success because it simplified and improved the teleoperation experience for the operator.

5.2 Future Work

While the results from testing indicated an improvement, some modifications can be implemented to improve the command strategy. There are limitations to the hardware of the Omnibot MMS that diminish the impact of the new command strategy. There is also additional hardware that can be used with the Omnibot MMS to improve performance and functionality. With these additions, the command strategy presented herein will allow operators to complete a variety of tasks quicker, more efficiently, with greater accuracy, and less concentration.

5.2.1 Hardware Improvements

5.2.1.1 Localization

The most important hardware improvement for the Omnibot MMS is to the localization system. Increased accuracy in localization will allow more precise tracking of the Omnibot MMS. With better localization, repulsive virtual fixtures can be used to help the operators avoid coming close to walls, reducing the need for the forbidden region virtual fixtures. The

issues regarding localization discussed in Section 4.3.1.3, which were the biggest setback for the new command strategy, will become nonexistent.

5.2.1.2 Drive Train

The drive train of the Omnibot has an inconsistent amount of backlash when driving. This can lead to a slight loss of orientation when going from stationary to moving. Since the Omnibot is controlled in world coordinates, this does not affect the operator but can sometimes put the manipulator in a less than optimal position.

5.2.1.3 Peripherals

One hardware subsystem that was not used in this work is the IR sensors mounted around the Omnibot. These sensors can be used to create repulsive virtual fixtures when near walls or obstacles. Since the goal of this work was to test the command strategy, creating virtual fixtures on the fly was not necessary to fulfill the goals of this research.

5.2.1.4 Manipulator

By adding an additional 3-DOF to the manipulator the operator will gain the ability to control the orientation of the manipulator's end-effector as well as its position. A gripper or a tool on the end-effector will increase the functionality of the MMS. This will require the use of a 6-DOF haptic device as well. Since these features will only improve the functionality of the Omnibot MMS, and not affect the command strategy they were not implemented on the current MMS.

5.2.1.5 Haptic Joystick

Having an additional "twist" DOF on the joystick would have improved the operators' ability to control the orientation of the Omnibot MMS. Instead, the operators were required to use two buttons located on the joystick's end-effector to control the rotation of the Omnibot. This meant that the rotation speed was fixed. Using the haptic joystick to control the rotation of the Omnibot, operators would be able to change the orientation at whatever speed they felt necessary.

5.2.2 Hardware Additions

5.2.2.1 3D Vision Systems

A 3D vision system would be an excellent addition to the Omnibot MMS. Using the vision system virtual fixtures can be created as needed, and not preprogrammed. With a vision system, attractive virtual fixtures (similar to the ones in section 3.4.1) can be created to assist the operator in the pick-and-place tasks. As well, the vision system can be used to determine if the pick-and-place task is complete, and switch from manipulation mode to transportation mode immediately, without requiring the operator to drive the Omnibot MMS away from the targets using near-target manipulation mode. A laser scanner or vision system can be used to create the repulsive virtual fixtures around the walls, and attractive virtual fixtures around the targets when in transportation mode.

5.2.2.2 Graphical User Interface

A Graphical User Interface (GUI) can help the operator in a variety of ways. One issue operators had when performing test runs is the manipulator blocked their view of the pick-and-place items. This made it very difficult to accurately perform the simulated pick-and-place

tasks. With a camera mounted on the Omnibot and a GUI displaying the camera's view, the operators would be able to see the pick-and-place items up close with an unobstructed view. The GUI can be used to compliment the virtual fixtures by displaying a visual representation of them. If the GUI is implemented through the use of 3D goggles, the virtual fixtures can be seen in 3D along with the Omnibot's environment.

5.2.2.3 Force/Torque Sensors

In order to make teleoperation more immersive, a force and torque sensor can be placed on the manipulator's end-effector. The sensor will allow the operator to feel the forces the manipulator is exerting on its environment allowing the operator to perform more delicate and precise tasks.

References

[1] Brad Hamner, Seth Koterba, Jane Shi, Reid Simmons, and Sanjiv Singh, "An autonomous mobile manipulator for assembly tasks," *Autonomous Robots*, 2010.

[2] M. Hvilshj and S. Bgh, "Little helper - an autonomous industrial mobile manipulator concept," *International Journal of Advanced Robotic Systems*, 2011.

[3] S. Datta, R. Ray, and D. Banerji, "Development of autonomous mobile robot with manipulator for manufacturing environment," *International Journal of Advanced Manufacturing Technology*, 2008.

[4] L. Tsai and S. Joshi, "Kinematics and optimization of a spatial 3-upu parallel manipulator," *Journal of Mechanical Design*, 2000.

[5] S. B. Nokleby and R. Podhorodeski, "A Complete Family of Kinematically-Simple Joint Layouts: Layout Models, Associated Displacement Problem Solutions and Applications," *Industrial Robotics Theory Modelling Control*, December 2006.

[6] C. V. Ferreira and V. F. Romano, "A design methodology for the compensation of positioning deviation in gantry manipulators," *Journal of the Brazilian Society of Mechanical Sciences*, 2002.

[7] A Morales, T Asfour, and D Osswald, "Towards an anthropomorphic manipulator for an assistant humanoid robot," *Robotics: Science and Systems - Workshop on Humanoid Manipulation*, 2006.

[8] M. Frejek, "Novel teleoperation of mobile-manipulator systems," M.S. thesis, University of Ontario: Institute of Technology, 2009.

[9] P. F. Hokayem and M. W. Spong, "Bilateral teleoperation: An historical survey," *Automatica*, 2006.

[10] J. Steuer, "Defining virtual reality: Dimensions determining telepresence," *Journal of Communication*, 1992.

[11] L.B. Rosenberg, "Virtual fixtures: Perceptual tools for telerobotic manipulation," *Virtual Reality Annual International Symposium*, 1993.

[12] M. Frejek and S. Nokleby, "Simplified tele-operation of mobile manipulator systems using knowledge of their singular configurations," in *ASME International Design Engineering Technical Conferences*, 2009, pp. 411–418.

[13] Jaeheung Park and Oussama Khatib, *Robust Haptic Teleoperation of a Mobile Manipulation Platform*, Springer Berlin / Heidelberg, 2006.

[14] Northrop Grumman, "Products," *http://www.is.northropgrumman.com/by_solution/remote_platforms*, 2011.

[15] iRobot, "irobot ground robots," *http://www.irobot.com/gi/ground/510_PackBot*, 2011.

[16] Quinetiq, "Talon robot," *http://www.qinetiq-na.com/products-talon.htm*, 2011.

[17] Dawei Wang, Jianqiang Yi, and Dongbin Zhao, "Teleoperation system of the internet-based omni-directional mobile robot with a mounted manipulator," in *Proceedings of the IEEE International Conference on Mechatronics and Automation*. IEEE, 2007, pp. 1799–1804.

[18] Palankar M., De Laurentis K.J., Alqasemi R., Veras E., Dubey R., Arbel Y., and Donchin E., "Control of a 9-dof wheelchair-mounted robotic arm system using a

p300 brain computer interface: Initial experiments," in *2008. ROBIO 2008. IEEE International Conference on Robotics and Biomimetics*. IEEE, 2009, pp. 348–353.

[19] B. Sicilliano, "Kinematic control of redundant robot manipulators: A tutorial," *Journal of Intelligent and Robotic Systems*, vol. 3, pp. 201–212, 1990.

[20] Whitney D. E., "Resolved motion rate control of manipulators and human protheses," *IEEE Trans. Man-Machine Systems*, vol. 10, pp. 47–53, 1969.

[21] Baillieul J., Hollerbach J. M., and Brockett R. W., "Programming and control of kinematically redundant manipulators," *Proc. 23rd IEEE Conf. Decision and Control*, pp. 768–774.

[22] Klein C. A. and Huang C. H., "Review of pseudoinverse control for use with kinematically redundant manipulators," *IEEE Trans. Systems Man Cybernet.*, vol. 13, pp. 245–250, 1983.

[23] Shamir T. and Yomdin Y., "Repeatability of redundant manipulators: Mathematical solution of the problem," *IEEE Trans. Automat. Control*, vol. 33, pp. 1004–1008, 1983.

[24] Sciavicco L. and Siciliano B., "A dynamic solution to the inverse kinematic problem for redundant manipulators," *Proc. 1987 IEEE lnternat. Conf. Robot. Automat., IEEE Computer Society Press, Washington*, pp. 1081–1087, 1987.

[25] Sciavicco L. and Siciliano B., "A solution algorithm to the inverse kinematic problem for redundant manipulators," *IEEE J. Robot. Automat.*, vol. 4, pp. 403–410, 1988.

[26] Chevallereau C. and Khalil W., "A new method for the solution of the inverse kinematics of redundant robots," *Proc. 1988 IEEE Internat. Conf. Robot. Automat., IEEE Computer Society Press, Washington*, pp. 37–42, 1988.

[27] Liegeois A., "Automatic supervisory control of the configuration and behavior of multibody mechanisms," *IEEE Trans. Systems Man Cybernet.*, vol. 7, pp. 868–871, 1977.

[28] Yoshikawa T., "Dynamic manipulability of robot manipulators," *J. Robot. Systems*, vol. 2, pp. 113–124, 1985.

[29] Yoshikawa T., "Manipulability of robotic mechanisms," *Internat. J. Robot. Res.*, vol. 4, pp. 3–9, 1985.

[30] Dubey R. V., Euler J. A., and Babcock S. M., "An efficient gradient projection optimization scheme for a seven-degree-of-freedom redundant robot with spherical wrist-proc. 1988 ieee internat. conf. robot. automat.," *IEEE Computer Society Press, Washington*, pp. 28–36, 1988.

[31] L. Sciavicco and B. Siciliano, "Solving the inverse kinematic problem for robotic manipulators," *in Proceedings of the 6th CISM-IFToMM Symposium Theory and Practice of Robots and Manipulators*, pp. 107–114, 1987.

[32] Baillieul J., "Kinematic programming alternatives for redundant manipulators," *Proc. 1985 IEEE Internat. Conf. Robot. Automat.*, vol. IEEE Computer Society Press, Silver Spring, pp. 722–728, 1985.

[33] Baker D. R. and Wampler C. W., "On the inverse kinematics of redundant manipulators," *lnternat. J. Robot. Res.*, vol. 7, pp. 3–21, 1988.

[34] Nakamura Y., Hanafusa H., and Yoshikawa T., "Task-priority based redundancy control of robot manipulators," *Internat. J. Robot. Res.*, vol. 6, pp. 3–15, 1987.

[35] Baillieul J., "Avoiding obstacles and resolving kinematic redundancy," *Proc. 1986 1EEE lnternat. Conf. Robot. Automat*, vol. IEEE Computer Society Press, Washington, pp. 1698–1704, 1986.

[36] T. Takubo, H. Arai, and K. Tanie, "Control of mobile manipultor using a virtual impedance wall," in *Proceedings of the 2002 IEEE International Conference on Robotics and Automation*. IEEE, 2002, pp. 3571 – 3576.

[37] Novint, "Novint - products," *http://www.novint.com/index.php/products*, 2011.

[38] Haption SA, "Haption virtuose 6d desktop," *http://www.haption.com/site/index.php/en/products-menu-en/hardware-menu-en/virtuose-6ddesktop-menu-en*, 2011.

[39] Sensable Technologies, "Phantom omni," *http://www.sensable.com/haptic-phantom-omni.htm*, 2011.

[40] E. S. Clapan and F. G. Hamza-Lup, "Simulation and training with haptic feedback a review," *International Conference on Virtual Learning*, 2008.

[41] C. Basdogan and C. Ho, "Principals of haptic rendering for virtual environments," *network.ku.edu.tr/˜cbasdogan/tutorials/haptic_tutorial.html*, 2002.

[42] F. G. Hamza-Lup and M. Adams, "Feel the pressure: e-learning system with haptic feedback," *16th Symposium on Haptic Interfaces for Virtual Environments and Teleoperator Systems*, 2008.

[43] C. K. Chui, J. S. K. Ong, Z. Y. Lian, Z. Wang, and J. Teo, "Haptics in computer-mediated simulation: Training in vertebroplasty surgery," *Simulation and Gaming*, pp. 438–451, 2006.

[44] S. A. Brewster, "The impact of haptic touching technology on cultural applications," *Proceedings of EVA2001*, pp. 1–14, 2001.

[45] C. Gunn and A. Mettenmeyer, "Virtual surgery across the world," *CSIRO Media Release*, 2002.

[46] R. G. Menedez and J. E. Bernard, "Advancing the state of the art in flight simulation via the use of synthetic environments," *Iowa Space Grant Consortium*, 2001.

[47] Handshake VR News, "Telehaptics for training and command and control," *The MSIAC's M&S Journal Online*, 2004.

[48] S. Wall, "An investigation of temporal and spatial limitations of haptic interfaces," *Department of Cybernetics, vol. Ph.D. Reading: University of Reading.*, 2004.

[49] J.C. Huegel and M.K. OMalley, "Progressive haptic and visual guidance for training in a virtual dynamic task," *IEEE Haptics Symposium 2010*, 2010.

[50] Ali Ghanbari, Hamid Abdi, Ben Horan, Saeid Nahavandi, Xiaoqi Chen, and Wenhui Wang, "Haptic guidance for microrobotic intracellular injection," *Proceedings of the 2010 3rd IEEE RAS & EMBS International Conference on Biomedical Robotics and Biomechatronics,*, 2010.

[51] L. Moody, C. Baber, and T. N. Arvanitis, "The role of haptic feedback in the training and assessment of surgeons," *Eurohaptics 2001, Birmingham, UK. University of Birmingham*, 2002.

[52] R.M. Williams II, M. Srivastava, R.R. Conatser, and J.N. Howell, "Implementation and evaluation of haptic playback system," *http://www.hapticse.org*, 2004.

[53] C. Basdogan, C. Ho, M.A. Srinivasan, and M. Slater, "An experimental study on the role of touch in shared virtual environments," *ACM Transactions of Computer-Human. Interaction, Vol.7*, 2000.

[54] C. Jay, M. Glencross, and R. Hubbold, "Modeling the effects of delayed haptic and visual feedback in a collaborative virtual environmen," *ACM Transactions of Computer-Human Interactions, Vol.14*, 2007.

[55] M.D. Elton and W.J. Book, "Comparison of human-machine interfaces designed for novices teleoperating multi-dof hydraulic manipulators," *IEEE International Symposium on Robot and Human Interactive Communication*, 2011.

[56] I. Farkhatdinov, J.H. Ryu, and J. An, "A preliminary experimental study on haptic teleoperation of mobile robot with variable force feedback gain," *IEEE Haptics Symposium 2010*, 2010.

[57] Shinsuk Park, Robert D. Howe, and David F. Torchiana, "Virtual fixtures for robotic cardiac surgery," *Proceedings of the 4th International Conference on Medical Image Computing and Computer-Assisted Intervention*, 2001.

[58] Mehdi Ammi and Antoine Ferreira, "Robotic assisted micromanipulation system using virtual fixtures and metaphors," *IEEE International Conference on Robotics and Automation*, 2007.

[59] Panadda Marayong, Gregory D. Hager, and Allison M. Okamura, "Control methods for guidance virtual fixtures in compliant human-machine interfaces," *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2008.

[60] Roland Tschakarow, "Powercube first steps: Mechanical and electrical connection, software installation, pc tools," *Amtec Robotics GmbH*, 2007.

[61] Roland Tschakarow, "Programmers guide for powercubes," *Amtec Robotics GmbH*, 2007.

[62] John J. Craig, *Introduction to Robotics: Mechanics and Control (3rd Edition)*, Pearson Education, 2004.

[63] S. Bemis, B. Reiss, and S. Nokleby, "Design and control of an omnibot autonomous vehicle," in *ASME International Design Engineering Technical Conferences*, 2008, pp. 877–884.

[64] S. Ginzburg, F. Frankenberg, and S. Nokleby, "Design and implementation of an indoor localizaion system for the omnibot omni-directional platform," in *Transactions of the Canadian Society for Mechanical Engineering*. CSME, 2009, pp. 715–729.

[65] M. Wrock and S. Nokleby, "Decoupled teleoperation of a holonomic mobile-manipulator system using automatic switching," *IEEE Canadian Conference on Electrical and Computer Engineering*, 2011.

[66] M. Wrock and S. Nokleby, "Command strategies for tele-operation of mobile-manipulator systems via a haptic input device," *ASME/IEEE International Conference on Mechatronic and Embedded Systems and Applications*, 2011.

[67] M. Wrock and S. Nokleby, "Haptic teleoperation of a manipulator using virtual fixtures and hybrid position-velocity control," *IFToMM World Congress in Mechanism and Machine Science*, 2011.

Appendix A

Testing Results

A.1 Virtual Wall vs. Optimal Manipulability Pose

no orientation

run	Total Time	Total Dista	Total time	Total dista	total time s	number of	time spent correcting
1	73.777	14.83032	36.4	3.647071	37.377	5	6.209
2	66.257	11.05217	29.826	3.031154	36.431	4	6.512
3	57.537	9.933365	28.416	2.608616	29.121	4	6.146
4	59.515	2733.483	29.978	3.055722	29.537	5	7.188
5	51.958	11.4625	25.571	2.441832	26.387	4	6.044

orientation

run	Total Time	Total Dista	Total time	Total dista	total time s	number of	time spent correcting
1	127.985	23.41121	70.104	4.03419	57.881	14	31.091
2	94.478	22.58739	50.752	3.895464	43.726	4	12.699
3	108.326	17.38316	56.827	4.008092	51.499	7	13.292
4	73.316	17.30053	39.121	2.954357	34.195	5	3.158
5	57.141	14.6835	30.831	2.846702	26.31	4	2.895

virtual wall

run	Total Time	Total Dista	Total time	Total dista	total time s	number of	time spent correcting
1	85.736	14.76917	34.181	3.998107	51.555	6	8.47
2	69.342	12.3656	30.673	2.922703	38.669	3	7.049
3	82.516	18.68256	37.288	3.076353	45.228	3	6.884
4	70.5	10.4941	30.16	2.695359	40.34	5	7.027
5	65.559	12.46414	28.906	2.608149	36.653	5	7.651

distance corrected	average tir	average bc	average bc	average ar	average ar	average nu	average tir
2.309440198	61.8088	556.1523	30.0382	2.956879	31.7706	4.4	6.4198
0.878040776	61.8088	556.1523	30.0382	2.956879	31.7706	4.4	6.4198
0.822211677	61.8088	556.1523	30.0382	2.956879	31.7706	4.4	6.4198
1.278480592	61.8088	556.1523	30.0382	2.956879	31.7706	4.4	6.4198
0.994708159	61.8088	556.1523	30.0382	2.956879	31.7706	4.4	6.4198

distance corrected	average tir	average bc	average bc	average ar	average ar	average nu	average tir
10.4844277	92.2492	19.07316	49.527	3.547761	42.7222	6.8	12.627
4.320537881	92.2492	19.07316	49.527	3.547761	42.7222	6.8	12.627
1.824279402	92.2492	19.07316	49.527	3.547761	42.7222	6.8	12.627
1.87366628	92.2492	19.07316	49.527	3.547761	42.7222	6.8	12.627
1.658352104	92.2492	19.07316	49.527	3.547761	42.7222	6.8	12.627

distance corrected	average tir	average bc	average bc	average ar	average ar	average nu	average tir
2.115347717	74.7306	13.75511	32.2416	3.060134	42.489	4.4	7.4162
1.010739618	74.7306	13.75511	32.2416	3.060134	42.489	4.4	7.4162
2.005033586	74.7306	13.75511	32.2416	3.060134	42.489	4.4	7.4162
1.016345255	74.7306	13.75511	32.2416	3.060134	42.489	4.4	7.4162
1.352050981	74.7306	13.75511	32.2416	3.060134	42.489	4.4	7.4162

average distance corrected
1.256576
1.256576
1.256576
1.256576
1.256576

average distance corrected
4.032253
4.032253
4.032253
4.032253
4.032253

average distance corrected
1.499903
1.499903
1.499903
1.499903
1.499903

Appendix B

Programming Documentation

B.1 Teleoperation Computer

B.1.1 Main.cpp

```
#include <math.h>
#include <cstdlib>
#include <iostream>
#include "haptics.h"
#include "m5apiw32.h"
#include <time.h>
#include <fstream>
#include <stdio.h>
#include <time.h>
#include <iostream>
#include <winsock2.h>
#include <string.h>
#include <windows.h>

#pragma comment(lib,"ws2_32.lib")

// The haptics object, with which we must interact
HapticsClass gHaptics;

const double gStiffness = 10;
const double gCubeEdgeLength = 1;
float a = 0.0;
float b = 0.228;
float d = 0.3125;
float omega1, omega2, omega3;
float theta1;
float theta2;
float theta3;


int main()
{
    using namespace std;
    char IPaddr[14]= "192.168.1.117";
    int port = 1121;

    char sendbuf[200];
    char recvbuf[200];
    int sockret = -1;
    int ret;

    int ctr;
    char Estate = 49;//1
    char Tstate, Rstate, Fstate, Vfix;
    double jtar[7];
    char * pch;
    float j1pos, j2pos, j3pos;
    int jctr;
    char tempc;
    double botX, botY, botTheta, worldTheta;
    float armX, armY, armZ;
    float Xlimit_max = 0.45;
    float Xlimit_min = 0.2;
    float Ylimit_max = 0.3;
    float Ylimit_min = -0.3;
```

```
float forcewall = 0.02;

//virtual walls
float Ywall = 1.15;
float YwallB = 2.7;
float XwallL = 2.1;
float XwallML = 1.45;
float XwallMR = 0.5;
float XwallR = 0;
float wallD = 0.1;
float wallScaling = 100;


float Vx, Vy, Vz;

clock_t tstamp;

ofstream data_log;
data_log.open("data_recv.csv");

// Call the haptics initialization function
gHaptics.init(gCubeEdgeLength, gStiffness);

gHaptics.centering_stiffness = 75;
gHaptics.Xworkspace_limit = 0;
gHaptics.Yworkspace_limit = 0;
gHaptics.Xfixture_force = 0;
gHaptics.Yfixture_force = 0;
gHaptics.fixreset = 0;

////////////////////////////////Socket Initialization//////////////////////////////
// Initialise Winsock
WSADATA WsaDat;
if(WSAStartup(MAKEWORD(2,2),&WsaDat)!=0)
{
    std::cout<<"Winsock error - Winsock initialization failed\r\n";
    WSACleanup();
    system("PAUSE");
    return 0;
}

    printf("initialized\n");

// Create our socket
SOCKET Socket=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
if(Socket==INVALID_SOCKET)
{
    std::cout<<"Winsock error - Socket creation Failed!\r\n";
    WSACleanup();
    system("PAUSE");
    return 0;
}

    printf("socket created\n");
// Resolve IP address for hostname
```

```
    struct hostent *host;

    /*if((host=gethostbyname("localhost"))==NULL)
    {
        std::cout<<"Failed to resolve hostname.\r\n";
        WSACleanup();
        system("PAUSE");
        return 0;
    }*/

    // Setup our socket address structure
    SOCKADDR_IN SockAddr;
    SockAddr.sin_port=htons(port);
    SockAddr.sin_family=AF_INET;
    SockAddr.sin_addr.s_addr=inet_addr(IPaddr);

    //SockAddr.sin_addr.s_addr=*((unsigned long*)host->h_addr);

        printf("connecting to socket\n");
    // Attempt to connect to server
    if(connect(Socket,(SOCKADDR*)(&SockAddr),sizeof(SockAddr))!=0)
    {

        int errret;
        errret = WSAGetLastError();     printf("fail\n");
        std::cout<<"Failed to establish connection with server\r\n";
        WSACleanup();
        system("PAUSE");
        return 0;
    }

    ////////////////////////////////////////////////////////////////

    // Display message from server
    char buffer[1000];
    memset(buffer,0,999);
    int inDataLength=recv(Socket,buffer,1000,0);
    std::cout<<buffer<<"\n";
    if (inDataLength == -1)         printf("Server: recv() error \n");
    printf("Connected: %d\n", inDataLength);
    ret = sprintf(sendbuf, "Hello Server");
    sockret = send(Socket, sendbuf, strlen(sendbuf), 0);
    printf("send() is ok: %d\n", sockret);
//////////Main loop starts here///////////////////////

Tstate = 33;
Rstate = 33;
Vfix = 41;// 41 -> Total teleoperation 42-> 2 joystick


    while(Estate == 49){

        gHaptics.getPosition();

        if(gHaptics.button_val == 4){//rear
```

```
    Fstate = 36;//$
    Estate = 48;//0

}
else if(gHaptics.button_val == 1){//center

    Fstate = 88;//X

}
else if(gHaptics.button_val == 2){//left

    Fstate = 76;//L

}
else if(gHaptics.button_val == 8){//right

    Fstate = 82;//R

}else{

    Fstate = 37;//%

}

Vx = gHaptics.cursorZ;
Vy = gHaptics.cursorX;
Vz = gHaptics.cursorY;

//Sending to Teleop MESSAGE 1
ret = sprintf(sendbuf, "%c%c%c%c%c#%f#%f#%f",Tstate,Fstate,Rstate,Estate,Vfix, Vx, Vy,
Vz);
printf("Sending message 1: %s\n", sendbuf);
sockret = send(Socket, sendbuf, strlen(sendbuf), 0);
printf("message sent, %d\n", sockret);

// Receive string from teleop MESSAGE 4
sockret = recv(Socket, recvbuf, 200, 0);

if (sockret == -1)

    printf("Server: recv() error \n");

else{

    printf("Received: %s\n", recvbuf);

    Tstate = recvbuf[0];
    Rstate = recvbuf[2];
    tstamp = clock();

    //Parse recived string

    pch = strtok(recvbuf, "#");
    jctr = 0;
```

```
while(pch != NULL){
    jtar[jctr] = atof(pch);
    printf("pch: %s\n",pch);
    jctr = jctr + 1;
    pch = strtok(NULL,"#");
    if(jctr > 6){
        break;
    }
}

botX = jtar[1];
printf("botX %f\n", botX);
printf("Tstate: %d\n", Tstate);

botY = jtar[2];

worldTheta = jtar[3];

armX = jtar[4];

armY = jtar[5];

armZ = jtar[6];
if(Vfix != 42){

    if(botY > 1.15 && (Tstate == 66 || Tstate == 68) && Vy < 0 && botX > 2.1){

        //gHaptics.Xfixture_force = 10;
        gHaptics.fixreset = 1;
        Vfix = 40;

    }else if(botY > 1.15 && (Tstate == 66 || Tstate == 68) && Vx > 0 && botX > 2.1){

        //gHaptics.Yfixture_force = 10;
        gHaptics.fixreset = 1;
        Vfix = 40;

    }else if(botY > 2.7 && (Tstate == 66 || Tstate == 68) && Vx > 0){

        //gHaptics.Yfixture_force = 10;
        gHaptics.fixreset = 1;
        Vfix = 40;

    }else if(botX < 0 && (Tstate == 66 || Tstate == 68) && Vy > 0){

        //gHaptics.Xfixture_force = -10;
        gHaptics.fixreset = 1;
        Vfix = 40;

    }else if(botY < 1.15 && (Tstate == 66 || Tstate == 68) && Vx < 0 && botX >0.5 &&
     botX < 1.45){

        //gHaptics.Yfixture_force = -10;
        gHaptics.fixreset = 1;
        Vfix = 40;
```

```
    }else if(botY < 1.15 && (Tstate == 66 || Tstate == 68) && Vy > 0 && botX > 1.0
    && botX < 1.45){

        //gHaptics.Xfixture_force = -10;
        gHaptics.fixreset = 1;
        Vfix = 40;

    }else if(botY < 1.15 && (Tstate == 66 || Tstate == 68) && Vy < 0 && botX > 0.5
    && botX < 1.0){

        //gHaptics.Xfixture_force = 10;
        gHaptics.fixreset = 1;
        Vfix = 40;

    }else{

        gHaptics.Yfixture_force = 0;
        gHaptics.Xfixture_force = 0;
        Vfix = 41;

    }
}

if(armX>(Xlimit_max - forcewall)){

    gHaptics.Xworkspace_limit = (((Xlimit_max-forcewall)-armX)*1000)*cos(worldTheta);
    gHaptics.Yworkspace_limit = (((Xlimit_max-forcewall)-armX)*1000)*sin(worldTheta);

}
else if(armX<(Xlimit_min + forcewall)){

    gHaptics.Xworkspace_limit = (-(armX-(Xlimit_min+forcewall))*1000)*cos(worldTheta
    );
    gHaptics.Yworkspace_limit = (-(armX-(Xlimit_min+forcewall))*1000)*sin(worldTheta
    );

}
else if(armY>(Ylimit_max - forcewall)){

    gHaptics.Xworkspace_limit = -(((Ylimit_max-forcewall)-armY)*1000)*sin(worldTheta
    );
    gHaptics.Yworkspace_limit = (((Ylimit_max-forcewall)-armY)*1000)*cos(worldTheta);

}
else if(armY<(Ylimit_min + forcewall)){

    gHaptics.Xworkspace_limit = -(-(armY-(Ylimit_min+forcewall))*1000)*sin(
    worldTheta);
    gHaptics.Yworkspace_limit = (-(armY-(Ylimit_min+forcewall))*1000)*cos(worldTheta
    );

}
else{
```

```
                gHaptics.Xworkspace_limit = 0;
                gHaptics.Yworkspace_limit = 0;

            }

            //record data to excel file
            data_log<<tstamp<<","<<botX<<","<<botY<<","<<worldTheta<<","<<armX<<","<<armY<<","<<
            armZ<<","<<Tstate<<","<<Fstate<<","<<Rstate<<","<<Vfix<<"\n";



        }




    }

    //close data recorder
    data_log.close();

    // Shutdown our socket
    shutdown(Socket,SD_SEND);

    // Close our socket entirely
    closesocket(Socket);

    // Cleanup Winsock
    WSACleanup();
    printf("Program Over :(\n");
    system("PAUSE");
    return 0;
}
```

B.1.2 Haptics.cpp

```
#include "haptics.h"
#include "m5apiw32.h"
#include <windows.h>
#include <math.h>

// Continuous servo callback function
HDLServoOpExitCode ContactCB(void* pUserData)
{
    // Get pointer to haptics object
    HapticsClass* haptics = static_cast< HapticsClass* >( pUserData );

    // Get current state of haptic device
    hdlToolPosition(haptics->m_positionServo);
    hdlToolButton(&(haptics->m_buttonServo));

    // Call the function that does the heavy duty calculations.
    haptics->cubeContact();

    // Send forces to device
    hdlSetToolForce(haptics->m_forceServo);

    // Make sure to continue processing
    return HDL_SERVOOP_CONTINUE;
}

// On-demand synchronization callback function
HDLServoOpExitCode GetStateCB(void* pUserData)
{
    // Get pointer to haptics object
    HapticsClass* haptics = static_cast< HapticsClass* >( pUserData );

    // Call the function that copies data between servo side
    // and client side
    haptics->synch();

    // Only do this once.  The application will decide when it
    // wants to do it again, and call CreateServoOp with
    // bBlocking = true
    return HDL_SERVOOP_EXIT;
}

// Constructor--just make sure needed variables are initialized.
HapticsClass::HapticsClass()
    : m_lastFace(FACE_NONE),
      m_deviceHandle(HDL_INVALID_HANDLE),
      m_servoOp(HDL_INVALID_HANDLE),
      m_cubeEdgeLength(1),
      m_cubeStiffness(1),
      m_inited(false)
{
    for (int i = 0; i < 3; i++)
        m_positionServo[i] = 0;
}

// Destructor--make sure devices are uninited.
```

```
HapticsClass::~HapticsClass()
{
    uninit();
}



void HapticsClass::init(double a_cubeSize, double a_stiffness)
{
    oldpos = 0;
    readyGO = true;
    cursorX = 0;
    cursorY = 0;
    cursorZ = 0;

    oldservo[0] = 0;
    oldservo[1] = 0;
    oldservo[2] = 0;

    PPstiffness = 500;

    m_cubeEdgeLength = a_cubeSize;
    m_cubeStiffness = a_stiffness;


    HDLError err = HDL_NO_ERROR;

    // Passing "DEFAULT" or 0 initializes the default device based on the
    // [DEFAULT] section of HDAL.INI.   The names of other sections of HDAL.INI
    // could be passed instead, allowing run-time control of different devices
    // or the same device with different parameters.  See HDAL.INI for details.
    m_deviceHandle = hdlInitNamedDevice("DEFAULT");
    testHDLError("hdlInitDevice");

    if (m_deviceHandle == HDL_INVALID_HANDLE)
    {
        MessageBox(NULL, "Could not open device", "Device Failure", MB_OK);
        exit(0);
    }

    // Now that the device is fully initialized, start the servo thread.
    // Failing to do this will result in a non-funtional haptics application.
    hdlStart();
    testHDLError("hdlStart");

    // Set up callback function
    m_servoOp = hdlCreateServoOp(ContactCB, this, bNonBlocking);
    if (m_servoOp == HDL_INVALID_HANDLE)
    {
        MessageBox(NULL, "Invalid servo op handle", "Device Failure", MB_OK);
    }
    testHDLError("hdlCreateServoOp");

    // Make the device current.  All subsequent calls will
    // be directed towards the current device.
```

```
    hdlMakeCurrent(m_deviceHandle);
    testHDLError("hdlMakeCurrent");

    // Get the extents of the device workspace.
    // Used to create the mapping between device and application coordinates.
    // Returned dimensions in the array are minx, miny, minz, maxx, maxy, maxz
    //                                      left, bottom, far, right, top, near)
    // Right-handed coordinates:
    //   left-right is the x-axis, right is greater than left
    //   bottom-top is the y-axis, top is greater than bottom
    //   near-far is the z-axis, near is greater than far
    // workspace center is (0,0,0)
    hdlDeviceWorkspace(m_workspaceDims);
    testHDLError("hdlDeviceWorkspace");


    // Establish the transformation from device space to app space
    // To keep things simple, we will define the app space units as
    // inches, and set the workspace to approximate the physical
    // workspace of the Falcon.  That is, a 4" cube centered on the
    // origin.  Note the Z axis values; this has the effect of
    // moving the origin of world coordinates toward the base of the
    // unit.
    double gameWorkspace[] = {-0.2,-0.2,-0.2,0.2,0.2,0.2};
    bool useUniformScale = true;
    hdluGenerateHapticToAppWorkspaceTransform(m_workspaceDims,
                                              gameWorkspace,
                                              useUniformScale,
                                              m_transformMat);
    testHDLError("hdluGenerateHapticToAppWorkspaceTransform");

    m_inited = true;

}

// uninit() undoes the setup in reverse order.  Note the setting of
// handles.  This prevents a problem if uninit() is called
// more than once.
void HapticsClass::uninit()
{
    if (m_servoOp != HDL_INVALID_HANDLE)
    {
        hdlDestroyServoOp(m_servoOp);
        m_servoOp = HDL_INVALID_HANDLE;
    }
    hdlStop();
    if (m_deviceHandle != HDL_INVALID_HANDLE)
    {
        hdlUninitDevice(m_deviceHandle);
        m_deviceHandle = HDL_INVALID_HANDLE;
    }
    m_inited = false;
}

// This is a simple function for testing error returns.  A production
```

```
// application would need to be more sophisticated than this.
void HapticsClass::testHDLError(const char* str)
{
    HDLError err = hdlGetError();
    if (err != HDL_NO_ERROR)
    {
        MessageBox(NULL, str, "HDAL ERROR", MB_OK);
        abort();
    }
}

// This is the entry point used by the application to synchronize
// data access to the device.  Using this function eliminates the
// need for the application to worry about threads.
void HapticsClass::synchFromServo()
{
    hdlCreateServoOp(GetStateCB, this, bBlocking);
}

// GetStateCB calls this function to do the actual data movement.
void HapticsClass::synch()
{
    // m_positionApp is set in cubeContact().
    m_buttonApp = m_buttonServo;
}

// A utility function to handle matrix multiplication.  A production application
// would have a full vector/matrix math library at its disposal, but this is a
// simplified example.
void HapticsClass::vecMultMatrix(double srcVec[3], double mat[16], double dstVec[3])
{
    dstVec[0] = mat[0] * srcVec[0]
        + mat[4] * srcVec[1]
        + mat[8] * srcVec[2]
        + mat[12];

    dstVec[1] = mat[1] * srcVec[0]
        + mat[5] * srcVec[1]
        + mat[9] * srcVec[2]
        + mat[13];

    dstVec[2] = mat[2] * srcVec[0]
        + mat[6] * srcVec[1]
        + mat[10] * srcVec[2]
        + mat[14];
}

// Here is where the heavy calculations are done.  This function is
// called from ContactCB to calculate the forces based on current
// cursor position and cube dimensions.  A simple spring model is
// used.
void HapticsClass::cubeContact()
{

    // Convert from device coordinates to application coordinates.
```

```
vecMultMatrix(m_positionServo, m_transformMat, m_positionApp);

m_forceServo[0] = 0;
m_forceServo[1] = 0;
m_forceServo[2] = 0;

// Skip the whole thing if not initialized
if (!m_inited) return;

double radiusWC = 0.0;
double pointLC[3];
double applyForce[6] = {0, 0, 0, 0, 0, 0};
int singularity_scaling = 10;
int proximity_scaling = 250;
double switching;
float test;

cursorX = m_positionApp[0];
cursorY = m_positionApp[1];
cursorZ = m_positionApp[2];

    m_forceServo[0] = -m_positionApp[0]*centering_stiffness - Yworkspace_limit -
    Xfixture_force;
    m_forceServo[1] = -m_positionApp[1]*centering_stiffness;
    m_forceServo[2] = -m_positionApp[2]*centering_stiffness - Xworkspace_limit -
    Yfixture_force;

//elimitate jitter
if((fabs(oldservo[0]-m_forceServo[0])>.75||fabs(oldservo[1]-m_forceServo[1])>.75||fabs(
oldservo[2]-m_forceServo[2])>.75)){

    if((Xfixture_force == 0 || Yfixture_force == 0) && fixreset == 1){

        fixreset = 0;

    }else{

        switching = m_forceServo[0];
        m_forceServo[0] = oldservo[0];
        oldservo[0] = switching;
        switching = m_forceServo[1];
        m_forceServo[1] = oldservo[1];
        oldservo[1] = switching;
        switching = m_forceServo[2];
        m_forceServo[2] = oldservo[2];
        oldservo[2] = switching;

    }

}else{

    oldservo[0] = m_forceServo[0];
    oldservo[1] = m_forceServo[1];
    oldservo[2] = m_forceServo[2];
```

```
    }


}

// Interface function to get current position
void HapticsClass::getPosition()
{
    cursorX = m_positionApp[0];
    cursorY = m_positionApp[1];
    cursorZ = m_positionApp[2];
    hdlToolButtons(&button_val);

}

// Interface function to get button state.  Only one button is used
// in this application.
bool HapticsClass::isButtonDown()
{
    hdlToolButton(&m_buttonApp);
    return m_buttonApp;
}

// For this application, the only device status of interest is the
// calibration status.  A different application may want to test for
// HDAL_UNINITIALIZED and/or HDAL_SERVO_NOT_STARTED
bool HapticsClass::isDeviceCalibrated()
{
    unsigned int state = hdlGetState();

    return ((state & HDAL_NOT_CALIBRATED) == 0);
}
```

B.1.3 Haptics.h

```
// Copyright 2007 Novint Technologies, Inc. All rights reserved.
// Available only under license from Novint Technologies, Inc.

// Make sure this header is included only once
#ifndef HAPTICS_H
#define HAPTICS_H

#include "hdl.h"
#include "hdlu.h"

// Know which face is in contact
enum RS_Face {
    FACE_NONE = -1,
    FACE_NEAR, FACE_RIGHT,
    FACE_TOP, FACE_FAR,
    FACE_LEFT, FACE_BOTTOM,
    FACE_DEFAULT,
    FACE_LAST               // reserved to allow iteration over faces
    };

// Blocking values
const bool bNonBlocking = false;
const bool bBlocking = true;

class HapticsClass
{

// Define callback functions as friends
friend HDLServoOpExitCode ContactCB(void *data);
friend HDLServoOpExitCode GetStateCB(void *data);

public:
    // Constructor
    HapticsClass();

    // Destructor
    ~HapticsClass();

    // Initialize
    void init(double a_cubeSize, double a_stiffness);

    // Clean up
    void uninit();

    // Get position
    void getPosition();

    // Get state of device button
    bool isButtonDown();

    // synchFromServo() is called from the application thread when it wants to exchange
    // data with the HapticClass object.  HDAL manages the thread synchronization
    // on behalf of the application.
    void synchFromServo();
```

```
    // Get ready state of device.
    bool isDeviceCalibrated();

    bool readyGO;
    float cursorX;
    float cursorY;
    float cursorZ;
    int state;
    double g_cubeStiffness;
    int complete;
    int button_val;
    int centering_stiffness;
    float Xworkspace_limit;
    float Yworkspace_limit;
    float Xfixture_force;
    float Yfixture_force;
    float fixreset;

private:
    // Move data between servo and app variables
    void synch();

    // Calculate contact force with cube
    void cubeContact();

    // Matrix multiply
    void vecMultMatrix(double srcVec[3], double mat[16], double dstVec[3]);

    // Check error result; display message and abort, if any
    void testHDLError(const char* str);

    // Nothing happens until initialization is done
    bool m_inited;

    // Transformation from Device coordinates to Application coordinates
    double m_transformMat[16];

    // Variables used only by servo thread
    double m_positionServo[3];
    bool   m_buttonServo;
    double m_forceServo[3];
    double oldservo[3];
    double att_obj_A[3];
    double att_obj_B[3];
    double rep_obj_A[3];

    // Variables used only by application thread
    double m_positionApp[3];
    bool   m_buttonApp;

    // Keep track of last face to have contact
    int    m_lastFace;

    // Handle to device
    HDLDeviceHandle m_deviceHandle;
```

```
    // Handle to Contact Callback
    HDLServoOpExitCode m_servoOp;

    // Device workspace dimensions
    double m_workspaceDims[6];

    // Size of cube
    double m_cubeEdgeLength;

    double m_cubeStiffness;

    //Variables for powercubes
    int ret;
    int modJ1;
    int modJ2;
    int modJ3;
    float j1Pos;
    float j2Pos;
    float j3Pos;
    float oldpos;
    float TcursorX;
    float TcursorY;
    float TcursorZ;
    int PPstiffness;
};

#endif  HAPTICS_H
```

B.2 Omnibot Laptop

B.2.1 Teleop.c

```c
#include <stdio.h>
#include <math.h>
#include "../include/m5apiw32.h"
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.h>
#include <stdlib.h>


void error(char *msg)
{
   // perror(msg);
}

int safe(void);
void joints(void);

int modJ1 = 16;
int modJ2 = 22;
int modJ3 = 24;
int vel = 1.5;
int acc = 2;
int dev = 0;
char Tstate, Fstate, Rstate, Vfix;
char Estate = 49; //Teleop, Falcon, Ros, End_Flag
float omega1, omega2, omega3;
float j1pos, j2pos, j3pos;
int ret;
float a = 0;
float b = 0.228;
float d = 0.3125;
float j1optimal = 0;
float j2optimal = 0;
float j3optimal = 0;
int retracted = 1;
float Xdist, Ydist, Dist, alpha, thetatar;
float Xoptimal = 0.28;
float Yoptimal = 0;
float pie = 3.14159265;
float mPose = 3.14159265;
float Xlimit = 1;
float Ylimit = 1;
float  theta1, theta2, theta3;
float Xlimit_max = 0.45;
float Xlimit_min = 0.2;
float Ylimit_max = 0.30;
float Ylimit_min = -0.30;
float forcewall = 0.02;
float armX, armY, armZ;
float Vx, Vy, Vz, theta;
float Vxlocal, Vylocal, Vzlocal;

int main(int argc, char **argv)
{
```

```
    int mode = 1; //(0 impedance wall, 1 retracting)
    int sockfd, newsockfd, portno;
    float j1postmp, j2postmp, j3postmp;
    socklen_t clilen;
    char buffer[256];
    struct sockaddr_in serv_addr, cli_addr;
    char lead=35;
    unsigned long serNo = 0;
    char strInitString[] = "ESD:0,250";
    int i;

//////////////////////////////////////////////////Server Code
(windows)/////////////////////////////////////////////

//Open Socket
     sockfd = socket(AF_INET, SOCK_STREAM, 0);
     if (sockfd < 0)
        error("ERROR opening socket");
     bzero((char *) &serv_addr, sizeof(serv_addr));

     portno = atoi(argv[1]); //Set port number

     serv_addr.sin_family = AF_INET;
     serv_addr.sin_addr.s_addr = INADDR_ANY;
     serv_addr.sin_port = htons(portno);
     if (bind(sockfd, (struct sockaddr *) &serv_addr,
              sizeof(serv_addr)) < 0)
              error("ERROR on binding");
//Listen for connection

printf("Waiting for connection\n");

     listen(sockfd,5);
     clilen = sizeof(cli_addr);
     newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
     if (newsockfd < 0)
          error("ERROR on accept");
     bzero(buffer,256);
 printf("Stopped Litening\n");
//////////////////////////////////////////////////////

int sockret = -1;
int ret;

char sendbuf[200] = "#0#0#0#";

// initialize to empty data...

char recvbuf[200] = "";



// Send acknowledge to client...
```

```
printf("Client Connected\n");


//printf("Float: %f\n", j1pos);
//set the send buffer data
ret = sprintf(sendbuf, "Hello Client 1");

sockret = send(newsockfd, sendbuf, strlen(sendbuf), 0);

if (sockret == -1){
       printf("Server: send() error \n");
    return 0;
}
else
{
       printf("send() is OK.\n");
}
    // Receive string from client

    sockret = recv(newsockfd, recvbuf, 200, 0);

    if (sockret == -1)

        printf("Server: recv() error \n");

    else{

        printf("Received: %s\n", recvbuf);

    }


     int sockfdros, newsockfdros, portnoros;
     socklen_t clilenros;
     char bufferros[256];
     struct sockaddr_in serv_addrros, cli_addrros;


//////////////////////////////////////////////////Server NUMBER TWO Code
(ros)//////////////////////////////////////////////

int sockfdr, newsockfdr, portnor, clilenr;
     struct sockaddr_in serv_addrr, cli_addrr;
     int nr;
     if (argc < 2) {
         fprintf(stderr,"ERROR, no port provided\n");
         exit(1);
     }
     sockfdr = socket(AF_INET, SOCK_STREAM, 0);
     if (sockfdr < 0)
        error("ERROR opening socket");
     bzero((char *) &serv_addrr, sizeof(serv_addrr));
     portno = atoi(argv[2]);
     serv_addrr.sin_family = AF_INET;
     serv_addrr.sin_addr.s_addr = INADDR_ANY;
```

```
     serv_addrr.sin_port = htons(portno);
     if (bind(sockfdr, (struct sockaddr *) &serv_addrr,
              sizeof(serv_addrr)) < 0)
              error("ERROR on binding");
     printf("Waiting for connection\n");
     listen(sockfdr,5);
     clilenr = sizeof(cli_addrr);
     newsockfdr = accept(sockfdr,
                 (struct sockaddr *) &cli_addrr,
                 &clilenr);
     if (newsockfdr < 0)
          error("ERROR on accept");
     printf("Client 2 connected \n");
     nr = write(newsockfdr,"Hello Client 2",14);
     if (nr < 0) error("ERROR writing to socket");
     printf("write() is ok\n");
     bzero(buffer,256);
     nr = read(newsockfdr,buffer,255);
     if (nr < 0) error("ERROR reading from socket");
     printf("Received: %s\n",buffer);
//////////////////////Powercubes Init////////////////////////////////////

//begin powercube initialization
printf("Initializing Powercube\n");

ret = PCube_openDevice( &dev, strInitString );
printf( "PCube_openDevice() returned: %d\n", ret);

//if succesful initialization, display modules found
if( ret == 0 )
{
    for( i = 1; i < MAX_MODULES; i++ )
    {
        ret = PCube_getModuleSerialNo( dev, i, &serNo );
        if( ret == 0 )
            printf( "Found module %d with SerialNo %d\n", i, serNo );


    }
}
else
{
    printf( "Unable to open Device! error %d\n", ret );
    return 0;
}

//reset powercubes
ret = PCube_resetAll(dev);

////////////////////////////////////////////////////

////////////////////////////////////Code Starts Here/////////////////////////////////////////
    double jtar[4]; //message 1: info#Vx#Vy#Vz
    char * pch;
    int jctr;
```

```
    double ptar[4]; //message 2: info#botX#botY#botTheta
    int pctr;
    float botX, botY, botTheta, worldTheta;

    Tstate = 65; //A, arm movement
    retracted = 1;

while(Estate == 49){

    joints();
    printf("J3: %f\n", j3pos);

    //printf("Waiting for Message 1:\n");
    //recieve joystick data from windows MESSAGE 1
    bzero(recvbuf, 200);
    sockret = recv(newsockfd, recvbuf, 200, 0);
    //printf("Received (message 1): %s\n", recvbuf);

    Fstate = recvbuf[1]; //Falcon state
    Estate = recvbuf[3]; //End_flag
    Vfix = recvbuf[4];
    printf("Vfix: %c\n",Vfix);
    //Parse recived string
    pch = strtok(recvbuf, "#");
    jctr = 0;
    while(pch != NULL){
        jtar[jctr] = atof(pch);
        jctr = jctr + 1;
        pch = strtok(NULL,"#");
    }

    Vxlocal = jtar[1];
    Vylocal = jtar[2];
    Vzlocal = jtar[3];


    //Receive pose from ROS MESSAGE 2
    bzero(buffer,256);
    //printf("Waiting for Message 2:\n");
    nr = read(newsockfdr, buffer,255);
    //printf("Received (Message 2): %s\n", buffer);
    Rstate = buffer[2];
    //Parse recived string
    pch = strtok(buffer, "#");
    pctr = 0;
    while(pch != NULL){
        ptar[pctr] = atof(pch);
        pctr = pctr + 1;
        pch = strtok(NULL,"#");
    }
    //store data to variables
    botX = ptar[1];
    botY = ptar[2];
    botTheta = ptar[3];
    //convert pose angle world frame
```

```
worldTheta = botTheta + mPose;\
//rollover pose angle
if(worldTheta > 6.28){
    worldTheta = worldTheta -6.28;
}
if(Rstate == 79 && Tstate == 67){
    Tstate = 68;
}

//Convert Velocity to world coordinates
Vx = Vxlocal * cos(worldTheta) + Vylocal * sin(worldTheta);
//printf("Vx %f\n",Vx);
Vy = -Vxlocal * sin(worldTheta) + Vylocal * cos(worldTheta);
Vz = Vzlocal;

//perform automatic mode switching
if(botY < 0.6 && mode == 1){
    mode = 0;
    Fstate = 88;
}else if(botY > 0.75 && mode == 0){
    mode = 1;
}

if(Vfix == 42){
    mode = 0;
}

//check for deadband condition
if(fabs(Vx) < 0.015 && fabs(Vy) < 0.015 && fabs(Vz) < 0.015){
    //printf("Deadband %d,%d,%d\n", Vx, Vy, Vz);
    Vx = 0;
    Vy = 0;
    Vz = 0;
    /////switch back from botmove
}

//if(Tstate == 66 && Fstate == 88){ //Tstate, B is Omnibot movement, Fstate, X is center
button
//  Tstate = 65; //switch back to A, arm movement
//}

//condition for impedance wall
if(Fstate == 88 || Fstate == 82 || Fstate == 76){
    Tstate = 66;
}else if(Tstate == 67 || Tstate == 68){}
else if(armX < Xlimit_min){

    if(Vx > 0){

        //drive omnibot
        Tstate = 87; //W Impedance wall

    }else if(Vx<0){

        Tstate = 65; //A arm operation
```

```
        }
    }else if(armX > Xlimit_max){

        if(Vx < 0){

            //drive omnibot
            Tstate = 87; //W Impedance wall

        }else if(Vx>0){

            Tstate = 65; //A arm operation

        }
    }else if(armY < Ylimit_min){

        if(Vy > 0){

            //drive omnibot
            Tstate = 87; //W Impedance wall

        }else if(Vy<0){

            Tstate = 65; //A arm operation

        }
    }else if(armY > Ylimit_max){

        if(Vy < 0){

            //drive omnibot
            Tstate = 87; //W Impedance wall

        }else if(Vy>0){

            Tstate = 65; //A arm operation

        }
    }else {

    Tstate = 65;

    }
    if(mode == 1 && j3pos > 1.16 && Tstate == 65){
        Tstate = 87;
    }

    //condition for automatic retracting

/*
    if(fabs(1.5707963267948966192313-j3pos) < 0.4 && j3pos != 0){
        //printf("retractings\n");
        lead = 82; //R, for retracting
    }
*/
```

```
if(Tstate == 65 && Vfix != 42){ //65(A) arm movement

    //perform jacobian transformation
    omega1 = (-sin(theta1)/(cos(theta2)*pow(cos(theta1),2)*b+cos(theta2)*pow(sin(theta1),2)*
    b+sin(theta2+theta3)*d*pow(cos(theta1),2)+pow(sin(theta1),2)*sin(theta2+theta3)*d))*Vx +
     (cos(theta1)/(cos(theta2)*pow(cos(theta1),2)*b+cos(theta2)*pow(sin(theta1),2)*b+sin(
    theta2+theta3)*d*pow(cos(theta1),2)+pow(sin(theta1),2)*sin(theta2+theta3)*d))*Vy;

    omega2 = (((-cos(theta1)*sin(theta2+theta3)*d-cos(theta1)*cos(theta2)*b+sin(theta1)*a)*
    sin(theta2+theta3))/(b*(pow(sin(theta2+theta3),2)*d*pow(cos(theta1),2)*sin(theta2)+sin(
    theta2+theta3)*d*pow(cos(theta1),2)*cos(theta2)*cos(theta2+theta3)+sin(theta2+theta3)*
    pow(cos(theta1),2)*cos(theta2)*b*sin(theta2)+pow(cos(theta1),2)*cos(theta2+theta3)*pow(
    cos(theta2),2)*b+pow(sin(theta2+theta3),2)*d*pow(sin(theta1),2)*sin(theta2)+sin(theta2+
    theta3)*d*pow(sin(theta1),2)*cos(theta2+theta3)*cos(theta2)+pow(sin(theta1),2)*cos(
    theta2)*b*sin(theta2+theta3)*sin(theta2)+pow(sin(theta1),2)*pow(cos(theta2),2)*b*cos(
    theta2+theta3))))*Vx + (-((sin(theta2+theta3)*sin(theta1)*d+sin(theta1)*cos(theta2)*b+
    cos(theta1)*a)*sin(theta2+theta3))/(b*(pow(sin(theta2+theta3),2)*d*pow(cos(theta1),2)*
    sin(theta2)+sin(theta2+theta3)*d*pow(cos(theta1),2)*cos(theta2)*cos(theta2+theta3)+sin(
    theta2+theta3)*pow(cos(theta1),2)*cos(theta2)*b*sin(theta2)+pow(cos(theta1),2)*cos(
    theta2+theta3)*pow(cos(theta2),2)*b+pow(sin(theta2+theta3),2)*d*pow(sin(theta1),2)*sin(
    theta2)+sin(theta2+theta3)*d*pow(sin(theta1),2)*cos(theta2+theta3)*cos(theta2)+pow(sin(
    theta1),2)*cos(theta2)*b*sin(theta2+theta3)*sin(theta2)+pow(sin(theta1),2)*pow(cos(
    theta2),2)*b*cos(theta2+theta3))))*Vy + (-(cos(theta2+theta3)/((sin(theta2+theta3)*sin(
    theta2)+cos(theta2+theta3)*cos(theta2))*b)))*Vz;

    omega3 = -((-((-cos(theta1)*sin(theta2+theta3)*d-cos(theta1)*cos(theta2)*b+sin(theta1)*a
    )/((cos(theta2)*pow(cos(theta1),2)*cos(theta2+theta3)+pow(sin(theta1),2)*cos(theta2+
    theta3)*cos(theta2)+sin(theta2+theta3)*pow(cos(theta1),2)*sin(theta2)+pow(sin(theta1),2
    )*sin(theta2+theta3)*sin(theta2))*d*b)))*Vx + ((sin(theta2+theta3)*sin(theta1)*d+sin(
    theta1)*cos(theta2)*b+cos(theta1)*a)/((cos(theta2)*pow(cos(theta1),2)*cos(theta2+theta3
    )+pow(sin(theta1),2)*cos(theta2+theta3)*cos(theta2)+sin(theta2+theta3)*pow(cos(theta1),2
    )*sin(theta2)+pow(sin(theta1),2)*sin(theta2+theta3)*sin(theta2))*d*b))*Vy + (-((sin(
    theta2)*b-cos(theta2+theta3)*d)/((sin(theta2+theta3)*sin(theta2)+cos(theta2+theta3)*cos(
    theta2))*d*b)))*Vz);
    //if motion is within operating limits, set joint speeds
    if(safe() == 1){

        ret = PCube_moveVel(dev, modJ1, omega1);
        ret = PCube_moveVel(dev, modJ2, omega2);
        ret = PCube_moveVel(dev, modJ3, omega3);
    }

}else if( Tstate == 87 && mode == 1){ //82(R) retracting

    Tstate = 67; //66(B) omnibot move
    //retract to maximum manipulability
    ret = PCube_moveRamp(dev, modJ1, j1optimal , vel, acc);
    ret = PCube_moveRamp(dev, modJ2, j2optimal , vel, acc);
    ret = PCube_moveRamp(dev, modJ3, j3optimal , vel, acc);
    retracted = 0;

    //calculate target orientation
    //distance from end effector to workspace center (X/Y optimal)
    Xdist = (armX - Xoptimal);
```

```
    Ydist = (armY - Yoptimal);
    Dist = sqrt(pow(Xdist,2) + pow(Ydist,2));
    printf("xdist %f\n ydist %f\n armx %f\n army %f\n", Xdist, Ydist, armX, armY);
    //angle of EE on XY plane from worspace center to EE
    alpha = atan2(Ydist,Xdist);
    printf("alpha: %f\n", alpha);
    //target orientation
    thetatar = (botTheta+alpha);
    if(thetatar < 0){
        thetatar = thetatar + 2*pie;
    }
    if(thetatar > 2*pie){
        thetatar = thetatar - 2*pie;
    }
    printf("thetatar: %f\n", thetatar);
}else if( Tstate == 87 && mode == 0){ //impedance wall

    Tstate = 66; //66(B) omnibot move
    ret = PCube_moveVel(dev, modJ1, 0);
    ret = PCube_moveVel(dev, modJ2, 0);
    ret = PCube_moveVel(dev, modJ3, 0);

}
else if(retracted == 1){
    ret = PCube_moveVel(dev, modJ1, 0);
    ret = PCube_moveVel(dev, modJ2, 0);
    ret = PCube_moveVel(dev, modJ3, 0);
}
if(Vfix == 42 && Fstate == 88){
    //perform jacobian transformation
    omega1 = (-sin(theta1)/(cos(theta2)*pow(cos(theta1),2)*b+cos(theta2)*pow(sin(theta1),2)*
    b+sin(theta2+theta3)*d*pow(cos(theta1),2)+pow(sin(theta1),2)*sin(theta2+theta3)*d))*Vx +
     (cos(theta1)/(cos(theta2)*pow(cos(theta1),2)*b+cos(theta2)*pow(sin(theta1),2)*b+sin(
    theta2+theta3)*d*pow(cos(theta1),2)+pow(sin(theta1),2)*sin(theta2+theta3)*d))*Vy;

    omega2 = (((-cos(theta1)*sin(theta2+theta3)*d-cos(theta1)*cos(theta2)*b+sin(theta1)*a)*
    sin(theta2+theta3))/(b*(pow(sin(theta2+theta3),2)*d*pow(cos(theta1),2)*sin(theta2)+sin(
    theta2+theta3)*d*pow(cos(theta1),2)*cos(theta2)*cos(theta2+theta3)+sin(theta2+theta3)*
    pow(cos(theta1),2)*cos(theta2)*b*sin(theta2)+pow(cos(theta1),2)*cos(theta2+theta3)*pow(
    cos(theta2),2)*b+pow(sin(theta2+theta3),2)*d*pow(sin(theta1),2)*sin(theta2)+sin(theta2+
    theta3)*d*pow(sin(theta1),2)*cos(theta2+theta3)*cos(theta2)+pow(sin(theta1),2)*cos(
    theta2)*b*sin(theta2+theta3)*sin(theta2)+pow(sin(theta1),2)*pow(cos(theta2),2)*b*cos(
    theta2+theta3))))*Vx + (-((sin(theta2+theta3)*sin(theta1)*d+sin(theta1)*cos(theta2)*b+
    cos(theta1)*a)*sin(theta2+theta3))/(b*(pow(sin(theta2+theta3),2)*d*pow(cos(theta1),2)*
    sin(theta2)+sin(theta2+theta3)*d*pow(cos(theta1),2)*cos(theta2)*cos(theta2+theta3)+sin(
    theta2+theta3)*pow(cos(theta1),2)*cos(theta2)*b*sin(theta2)+pow(cos(theta1),2)*cos(
    theta2+theta3)*pow(cos(theta2),2)*b+pow(sin(theta2+theta3),2)*d*pow(sin(theta1),2)*sin(
    theta2)+sin(theta2+theta3)*d*pow(sin(theta1),2)*cos(theta2+theta3)*cos(theta2)+pow(sin(
    theta1),2)*cos(theta2)*b*sin(theta2+theta3)*sin(theta2)+pow(sin(theta1),2)*pow(cos(
    theta2),2)*b*cos(theta2+theta3))))*Vy + (-(cos(theta2+theta3)/((sin(theta2+theta3)*sin(
    theta2)+cos(theta2+theta3)*cos(theta2))*b)))*Vz;

    omega3 = -((-((-cos(theta1)*sin(theta2+theta3)*d-cos(theta1)*cos(theta2)*b+sin(theta1)*a
    )/((cos(theta2)*pow(cos(theta1),2)*cos(theta2+theta3)+pow(sin(theta1),2)*cos(theta2+
    theta3)*cos(theta2)+sin(theta2+theta3)*pow(cos(theta1),2)*sin(theta2)+pow(sin(theta1),2
```

```
    )*sin(theta2+theta3)*sin(theta2))*d*b)))*Vx + ((sin(theta2+theta3)*sin(theta1)*d+sin(
    theta1)*cos(theta2)*b+cos(theta1)*a)/((cos(theta2)*pow(cos(theta1),2)*cos(theta2+theta3
    )+pow(sin(theta1),2)*cos(theta2+theta3)*cos(theta2)+sin(theta2+theta3)*pow(cos(theta1),2
    )*sin(theta2)+pow(sin(theta1),2)*sin(theta2+theta3)*sin(theta2))*d*b))*Vy + (-((sin(
    theta2)*b-cos(theta2+theta3)*d)/((sin(theta2+theta3)*sin(theta2)+cos(theta2+theta3)*cos(
    theta2))*d*b)))*Vz);
    //if motion is within operating limits, set joint speeds
    if(safe() == 1){

        ret = PCube_moveVel(dev, modJ1, omega1);
        ret = PCube_moveVel(dev, modJ2, omega2);
        ret = PCube_moveVel(dev, modJ3, omega3);
    }
}

if(retracted == 0){

    //printf("waiting for retract\n");
    ret = PCube_getPos( dev, modJ1, &j1postmp);
    ret = PCube_getPos( dev, modJ2, &j2postmp);
    ret = PCube_getPos( dev, modJ3, &j3postmp);

    if((fabs(j1postmp - j1optimal) + fabs(j2postmp - j2optimal) + fabs(j3postmp - j3optimal
    )) < 0.1){
        retracted = 1;
    }
}

if(retracted == 0){
    Vx = 0;
    Vy = 0;
    Vz = 0;
}

//populate send buffer for ROS MESSAGE 3
bzero(sendbuf, 200);
ret = sprintf(sendbuf, "%c%c%c%c%c#%f#%f#%f#%f#%f#%f#%f",Tstate, Fstate, Rstate, Estate,
Vfix, Vx, Vy, Vz, armX, armY, armZ, thetatar);
//printf("Waiting for Message 3: %s\n",sendbuf);
//send joystick data to ROS
nr = write(newsockfdr,sendbuf,200);
    if (nr < 0) error("ERROR writing to socket");
printf("sent Message 3: %c%c%c%c%c#%f#%f#%f#%f#%f#%f#%f\n",Tstate, Fstate, Rstate, Estate,
Vfix, Vx, Vy, Vz, armX, armY, armZ, thetatar);
//Send Pose to windows MESSAGE 4
bzero(sendbuf, 200);
ret = sprintf(sendbuf, "%c%c%c%c#%f#%f#%f#%f#%f#%f",Tstate, Fstate, Rstate, Estate, botX,
botY, worldTheta, armX, armY, armZ); //ptar is pose data
//printf("Waiting for Message 4: %s\n",sendbuf);
//printf("sending to windows: %s\n", sendbuf);
sockret = send(newsockfd, sendbuf, strlen(sendbuf), 0);
printf("Sent Message 4:%c%c%c%c#%f#%f#%f#%f#%f#%f\n",Tstate, Fstate, Rstate, Estate, botX,
botY, worldTheta, armX, armY, armZ);

}
```

```
    close(newsockfd);
    close(newsockfdr);
    printf("Program Over :(\n");
     return 0;
}

int safe(void){

    float tartheta1, tartheta2, tartheta3;
    float j1temp, j2temp, j3temp, j3pos2;

    joints();

    j1temp = j1pos + omega1/20;

    j2temp = j2pos + omega2/20;

    j3pos2 = j3pos + 1.570796326794896619 23;

    j3temp = j3pos2 + omega3/20;



    if(j1pos > 2.5 && omega1 > 0){

        ret = PCube_moveVel(dev, modJ1, 0);
        ret = PCube_moveVel(dev, modJ2, 0);
        ret = PCube_moveVel(dev, modJ3, 0);
        printf("joint limit error\n");

        return 0;
    }else if(j1pos < -2.5 && omega1 < 0){

        ret = PCube_moveVel(dev, modJ1, 0);
        ret = PCube_moveVel(dev, modJ2, 0);
        ret = PCube_moveVel(dev, modJ3, 0);
        printf("joint limit error\n");

        return 0;
    }else if(j2pos > 2.5 && omega2 > 0){

        ret = PCube_moveVel(dev, modJ1, 0);
        ret = PCube_moveVel(dev, modJ2, 0);
        ret = PCube_moveVel(dev, modJ3, 0);
        printf("joint limit error\n");

        return 0;
    }else if(j2pos < -2.5 && omega2 < 0){

        ret = PCube_moveVel(dev, modJ1, 0);
        ret = PCube_moveVel(dev, modJ2, 0);
        ret = PCube_moveVel(dev, modJ3, 0);
        printf("joint limit error\n");

        return 0;
```

```
}else if(j3pos > 2.5 && omega3 > 0){

    ret = PCube_moveVel(dev, modJ1, 0);
    ret = PCube_moveVel(dev, modJ2, 0);
    ret = PCube_moveVel(dev, modJ3, 0);
    printf("joint limit error\n");

    return 0;
}else if(j3pos < -2.35 && omega3 < 0){

    ret = PCube_moveVel(dev, modJ1, 0);
    ret = PCube_moveVel(dev, modJ2, 0);
    ret = PCube_moveVel(dev, modJ3, 0);
    printf("joint limit error\n");

    return 0;
}else if(omega1 > 2 || omega1 < -2 || omega2 > 2 || omega2 < -2 || omega3 > 2 || omega3 < -2
){

    ret = PCube_moveVel(dev, modJ1, 0);
    ret = PCube_moveVel(dev, modJ2, 0);
    ret = PCube_moveVel(dev, modJ3, 0);
    printf("joint velocity error\n");

    return 0;
}else if(j3pos > 1.15 && omega3 < 0 && j3pos2 > 0){

    ret = PCube_moveVel(dev, modJ1, 0);
    ret = PCube_moveVel(dev, modJ2, 0);
    ret = PCube_moveVel(dev, modJ3, 0);
    printf("joint singularity error\n");

    return 0;
}else if(j3pos2 > -0.10 && j3pos2 < 0 && omega3 > 0){

    ret = PCube_moveVel(dev, modJ1, 0);
    ret = PCube_moveVel(dev, modJ2, 0);
    ret = PCube_moveVel(dev, modJ3, 0);
    printf("joint singularity error\n");

    return 0;
}else if(retracted == 0){

    printf("not retracted\n");
    return 0;

}else if(armX>(Xlimit_max) && Vx<0){
    printf("not within xmax workspace\n");
    return 0;

}else if(armX<(Xlimit_min) && Vx>0){
    printf("not within xmin workspace\n");
    return 0;

}else if(armY>(Ylimit_max) && Vy<0){
```

```
        printf("not within ymax workspace\n");
        return 0;

    }else if(armY<(Ylimit_min) && Vy>0){
        printf("not within ymin workspace\n");
        return 0;

    }else{

        return 1;
    }

}
void joints(void){

    //Retrieve joint positions

    ret = PCube_getPos( dev, modJ1, &j1pos);
    ret = PCube_getPos( dev, modJ2, &j2pos);
    ret = PCube_getPos( dev, modJ3, &j3pos);

    //printf("j1 %f   j2 %f   j3 %f\n", j1pos, j2pos, j3pos);


    //jacobian uses a slightly different zero displacement
    theta1 = j1pos;
    theta2 = j2pos - 1.57079632679489661923;
    theta3 = -j3pos;

    //Compensate for zero-displacement configuration
    j2pos = -j2pos;
    j3pos = -j3pos;

    //rollover joint values
    if(j3pos > 3.14159265){
        j3pos = j3pos - 3.14159265;
    }
    if(j3pos < -3.14159265){
        j3pos = j3pos + 3.14159265;
    }
    if(j2pos > 3.14159265){
        j2pos = j2pos - 3.14159265;
    }
    if(j2pos < -3.14159265){
        j2pos = j2pos + 3.14159265;
    }
    if(theta2 > 3.14159265){
        theta2  = theta2  - 3.14159265;
    }
    if(theta2 < -3.14159265){
        theta2  = theta2  + 3.14159265;
    }

    //calculate forward displacement
    armX = cos(j1pos)*(b*sin(j2pos)+d*cos(j3pos-j2pos));
```

```
    armY = sin(j1pos)*(b*sin(j2pos)+d*cos(j3pos-j2pos));
    armZ = b*cos(j2pos)+d*sin(j3pos-j2pos);
//  printf("armX: %f\narmY: %f\narmZ: %f\n",armX,armY,armZ);

}
```

B.2.2 CubeListener_automaticSwitching.cpp

```
// ROS header files
#include <ros/ros.h>
#include <std_msgs/String.h>

#include <sstream>
#include <iostream>

#include <stdio.h>   /* Standard input/output definitions */
#include <string.h>  /* String function definitions */
#include <unistd.h>  /* UNIX standard function definitions */
#include <fcntl.h>   /* File control definitions */
#include <errno.h>   /* Error number definitions */
#include <termios.h> /* POSIX terminal control definitions */

#include <math.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <netdb.h>



/*
* 'open_port()' - Open serial port 1.
*
* Returns the file descriptor on success or -1 on error.
*/

int fd;
char posedata[50];
char joystring[50];
char command[50];
float botTheta, worldTheta;
double botX, botY;
char * pch;
int jctr, pctr;
double jtar[8], ptar[4];
char buffer[256];

struct termios options;

void error(char *msg)
{
    perror(msg);
}


void cubeCallback(const std_msgs::String::ConstPtr& msg)
{
    const char * inputMsg = msg->data.c_str();  // store the published msg in inputMsg
    char JoyStickData[19];
    int len, wr, len2;
    //ROS_INFO("I heard: [%s]", msg->data.c_str());
    sprintf(posedata, "%s\n",inputMsg);
```

```
    //parse pose data
    pch = strtok(posedata, ",");
    pctr = 0;
    while(pch != NULL){
        ptar[pctr] = atof(pch);
        pctr = pctr + 1;
        pch = strtok(NULL,",");
    }
    //store pose
    botX = ptar[0]/10000;
    botY = ptar[1]/10000;
    //convert pose angle to radians
    botTheta = ptar[2]/10000;
}


int main(int argc, char **argv)
{
int sockfd, portno, n, ret;
    struct sockaddr_in serv_addr;
    struct hostent *server;
double jtrans[2];
int robotx, roboty;
int botrotation = 128;
int botmove = 0;
float Xdist, Ydist, Dist, alpha, beta, posetarX, posetarY, thetatar;
float Xoptimal = -0.315;
float Yoptimal = 0.1413;
float pie = 3.14159265;
int oriented;
char Tstate, Rstate, Fstate, Vfix;
char Estate = 49;//1
float Vx, Vy, Vz;
float armX, armY, armZ;

////////////////////////////////ROS subscriptions//////////////////////////////
    // Declare this program as a ROS node, with the name "odometry" and NodeHandle n
    ros::init(argc, argv, "CubeListener");
    ros::NodeHandle pcube;
    // Subscribe to topic "cricket_data" to receive msgs from the cricket node. When msgs are
    available
    // on this topic the funtion cricketCallback is called to read in those msgs.
    ros::Subscriber sub = pcube.subscribe("pose_estimates", 1000, cubeCallback);

    // Publish to topic joyChatter
    ros::Publisher pubcom = pcube.advertise<std_msgs::String>("joyChatter", 1000);

    // Publish to topic user_commands
    //ros::Publisher pubcomm = pcube.advertise<std_msgs::String>("user_commands", 1000);

////////////////////////////////Connect to server/////////////////////////////////////
    if (argc < 3) {
       fprintf(stderr,"usage %s hostname port\n", argv[0]);
       exit(0);
    }
```

```
    portno = atoi(argv[2]);
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        error("ERROR opening socket");
    server = gethostbyname(argv[1]);
    if (server == NULL) {
        fprintf(stderr,"ERROR, no such host\n");
        exit(0);
    }
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    bcopy((char *)server->h_addr,
         (char *)&serv_addr.sin_addr.s_addr,
         server->h_length);
    serv_addr.sin_port = htons(portno);
    if (connect(sockfd,(sockaddr*) (&serv_addr),sizeof(serv_addr)) < 0)
        error("ERROR connecting");
    n = read(sockfd,buffer,255);
    if (n < 0)
         error("ERROR reading from socket");
    printf("Recieved: %s\n",buffer);
    ret = sprintf(buffer, "Hello Server");
    n = write(sockfd,buffer,strlen(buffer));
    if (n < 0)
         error("ERROR writing to socket");
    bzero(buffer,256);

////////////////////////////////////////


Rstate = 79; //O for ok
Tstate = 33;
Fstate = 33;
oriented = 0;

while(Estate == 49){

    ros::spinOnce();

    //send ros pose to teleop MESSAGE 2
    bzero(buffer,256);
    ret = sprintf(buffer, "%c%c%c%c#%f#%f#%f", Tstate, Fstate, Rstate, Estate, botX, botY,
    botTheta);
    printf("sending Message 2: %s\n",buffer);
    n = write(sockfd,buffer,strlen(buffer));

    //listen for teleop MESSAGE 3
    bzero(buffer,256);
    printf("Waiting for Message 3:\n");
    n = read(sockfd,buffer,255);
    printf("Received (message 3): %s\n",buffer);

    Tstate = buffer[0];
    Fstate = buffer[1];
    Estate = buffer[3];
```

```
Vfix = buffer[4];

//parse joystick data
pch = strtok(buffer, "#");
jctr = 0;
while(pch != NULL){
    jtar[jctr] = atof(pch);
    jctr = jctr + 1;
    pch = strtok(NULL,"#");
}

Vy = -jtar[1];

Vx = -jtar[2];

Vz = jtar[3];

armX = jtar[4];

armY = jtar[5];

armZ = jtar[6];

thetatar = jtar[7];

if(Tstate == 67){
    Rstate = 78;
}

if((Tstate == 66 || Tstate == 67 || Tstate == 68) && Vfix != 40){ //66(B) omnibot drive

    if(Fstate == 82){  //82(R)

        botrotation = 158;

    }else if(Fstate == 76){ //76(L)

        botrotation = 98;

    }else{

        botrotation = 128;
    }

    //switch from world to local coordinate frame
    worldTheta = botTheta + 3.14/2;// Change this to change world frame orientation
    printf("Vx: %f\nVy: %f\n", Vx, Vy);
    //convert from falcon scale (-0.2,0.2) to omnibot scale (0,255)
    jtrans[0] = (Vx + 0.2)*637.5;
    jtrans[1] = 255 - (Vy + 0.2)*637.5;
    //ensure values are within limits
    if(jtrans[0]>255) jtrans[0] = 255;
    if(jtrans[0]<0) jtrans[0] = 0;
    if(jtrans[1]>255) jtrans[1] = 255;
    if(jtrans[1]<0) jtrans[1] = 0;
```

```
    //cast values from float to int
    roboty = (int)(jtrans[0]);
    robotx = (int)(jtrans[1]);
    //botmove is equivalent to the deadman's switch for joylistener
    botmove = 1;
    //work on achieveing "snapshot" orientation
    printf("thetatar %f\noriented %d\n", thetatar, oriented);
    if(Rstate == 78){
        /*if(botTheta > (thetatar + 0.1)){
            botrotation = 153;
            printf("spin right\n");
        }else if(botTheta < (thetatar - 0.1)){
            botrotation = 113;
            printf("spin left\n");
        }else{*/
            botrotation = 128;
            oriented = 1;
            Rstate = 79;
        //}
    }
    if(Vfix != 40 && Vfix != 42){
    printf("I dont care about vfix: %d\n",Vfix);
    //send motion command to joylistener
    ret = sprintf(joystring, "%.3i,%.3i,%.3i,%i", robotx, roboty, botrotation, botmove);
    //ret = sprintf(joystring, "%3d,%3d,%3d,%d\n", robotx, roboty, botrotation, botmove);
    printf("sending: %.3i,%.3i,%.3i,%i\n", robotx, roboty, botrotation, botmove);
    //printf(joystring);
    // convert message to string for ROS node communication
    std_msgs::String msg;
    std::stringstream ss;
    ss << joystring;
    ROS_INFO("%s", ss.str().c_str());
    msg.data = ss.str();
    // publish data under topic joyChatter
    pubcom.publish(msg);

    ros::spinOnce();
    }
}else if(Tstate == 87){ //87(W) impedance wall

    botrotation = 128;
    jtrans[0] = (Vx + 0.2)*637.5;
    jtrans[1] = 255 - (Vy + 0.2)*637.5;
    //ensure values are within limits
    if(jtrans[0]>255) jtrans[0] = 255;
    if(jtrans[0]<0) jtrans[0] = 0;
    if(jtrans[1]>255) jtrans[1] = 255;
    if(jtrans[1]<0) jtrans[1] = 0;
    //cast values from float to int
    roboty = (int)(jtrans[0]);
    robotx = (int)(jtrans[1]);
    if(Vfix != 42){
        //botmove is equivalent to the deadman's switch for joylistener
        botmove = 1;
        //send motion command to joylistener
```

```
            ret = sprintf(joystring, "%.3i,%.3i,%.3i,%i", robotx, roboty, botrotation, botmove);
            printf("sending: %.3i,%.3i,%.3i,%i\n", robotx, roboty, botrotation, botmove);
            // convert message to string for ROS node communication
            std_msgs::String msg;
            std::stringstream ss;
            ss << joystring;
            ROS_INFO("%s", ss.str().c_str());
            msg.data = ss.str();
            // publish data under topic joyChatter
            pubcom.publish(msg);

            ros::spinOnce();
        }

    }/*else if(Tstate == 65 && (Fstate == 82 || Fstate == 76)){ //66(A) arm drive

        if(Fstate == 82){  //82(R)

            botrotation = 158;

        }else if(Fstate == 76){ //76(L)

            botrotation = 98;

        }else{

            botrotation = 128;
        }

        //send motion command to joylistener
        ret = sprintf(joystring, "128,128,%.3i,1",botrotation);
        //ret = sprintf(joystring, "%3d,%3d,%3d,%d\n", robotx, roboty, botrotation, botmove);
        printf("sending: %.3i,%.3i,%.3i,%i\n", robotx, roboty, botrotation, botmove);
        //printf(joystring);
        // convert message to string for ROS node communication
        std_msgs::String msg;
        std::stringstream ss;
        ss << joystring;
        ROS_INFO("%s", ss.str().c_str());
        msg.data = ss.str();
        // publish data under topic joyChatter
        pubcom.publish(msg);

        ros::spinOnce();

    }*/else if(Tstate == 81){ //Q crawling
/*
        //distance from end effector to workspace center (X/Y optimal)
        Xdist = -(jtar[3] - Xoptimal);
        Ydist = (jtar[4] - Yoptimal);
        Dist = sqrt(pow(Xdist,2) + pow(Ydist,2));
        //angle of EE on XY plane from worspace center to EE
        alpha = atan2(Ydist,Xdist);
        //angle from EE to world coordinates
        beta = theta - 0.75*pie - alpha;
```

```
        //target pose, the EE location
        posetarX = posex + Dist*cos(beta);
        posetarY = posey + Dist*sin(beta);
        //target orientation
        thetatar = (theta-alpha)*10000;
        if(thetatar < 0){
            thetatar = thetatar + 20000*pie;
        }
        if(thetatar > pie*20000){
            thetatar = thetatar - 20000*pie;
        }

        //send motion command to Pathfollowing
        ret = sprintf(command, "%d,%d,%d", (int)posetarX, (int)posetarY, (int)thetatar);
        printf("sending to pathfollowing: %d,%d,%d\n", (int)posetarX, (int)posetarY,
        (int)thetatar);

        // convert message to string for ROS node communication
        std_msgs::String msg;
        std::stringstream ss;
        ss << command;
        //ROS_INFO("%s", ss.str().c_str());
        msg.data = ss.str();
        // publish data under topic commands
        pubcomm.publish(msg);

*/
    }else if(Vfix != 42){

        botmove = 0;
        oriented = 0;
        //send motion command to joylistener
        ret = sprintf(joystring, "%.3i,%.3i,%.3i,%i", robotx, roboty, botrotation, botmove);
        //ret = sprintf(joystring, "%3d,%3d,%3d,%d\n", robotx, roboty, botrotation, botmove);
        printf("sending: %.3i,%.3i,%.3i,%i\n", robotx, roboty, botrotation, botmove);
        //printf(joystring);
        // convert message to string for ROS node communication
        std_msgs::String msg;
        std::stringstream ss;
        ss << joystring;
        ROS_INFO("%s", ss.str().c_str());
        msg.data = ss.str();
        // publish data under topic joyChatter
        pubcom.publish(msg);

        ros::spinOnce();

    }



}
```

```
    return 0;
}
```